

# **REFERENCE GUIDE**

# **GFA-BASIC PC**

**MS-DOS 80x86**



All rights reserved. No part of this handbook may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of GFA Systemtechnik GmbH, except for the personal use of the buyer.

The publisher has made every effort to publish the complete and accurate information. GFA Systemtechnik assumes no responsibility that the described procedures, programs etc. are functional and free from third party rights.

Program authors: Frank Ostrowski  
Roland Schütz  
Dirk van Assche  
Michael Wiegand

Handbook authors: Dr. Wolfgang Buscher  
Frank Ostrowski

Translation: Donald P. Maple

© Copyright 1990: GFA Systemtechnik GmbH

1. Edition, December 1990

# **Reference Handbook**

## **Reference**

**Appendix A:** GFA-BASIC editor

**Appendix B:** PC/XT and AT MF 2 Keyboard Scan Codes

**Code table:** Scan Codes

**Code table:** ASCII Character Set

**Code table:** IBM Character Set

**Appendix C:** DOS Interrupts

**Appendix D:** International sorting sequence

**Conversion:** hexadecimal - decimal



# Foreword

## GFA-BASIC PC

### A fine piece of work!

After only one year of development a team of four programmers, Frank Ostrowski, Dirk van Assche, Roland Schütz and Michael Wiegand, has managed to port the best selling programming language on the Atari ST to the PC. This is extraordinary on five counts:

1. The GFA-BASIC for the Atari ST is written completely in 68000 assembler. Porting it to the PC therefore means a change over to a completely different Intel architecture.
2. As on the Atari ST all executive parts of the GFA-BASIC are programmed fully in assembler. This means that only the Editor and its parser are written in C.
3. The GFA-BASIC PC programs are fine tuned to various processors in the Intel family. This means that, for example, the 8086/80286 versions of GFA-BASIC use 16 bit wide registers, while the 80386 version of GFA-BASIC uses 32 bit wide registers. The speed improvements achieved with this are enormous.
4. GFA-BASIC PC is, at this time, available for IBM compatible PCs under four operating systems:

GFA-BASIC for MS-DOS

GFA-BASIC for Windows 3.0

GFA-BASIC for OS/2 1.2

GFA-BASIX for SCO Open Desktop

5. "The highest possible syntax compatibility was observed during the creation of all versions. To realise this, a complete library for the management of a graphic user interface was built for the MS-DOS version. This includes menu bars, windows, alert boxes and pop-up menus. When creating these routines we observed the SAA standard. The command `OPENW #1,0,0,300,200,-1` creates therefore a graphic window in all GFA-BASIC versions. Whether this is done through GEM, GFA-BASIC's own, WINDOWS, OS/2 or X-Windows/Motif routines depends solely on which operating system you use on your computer and which version of GFA-BASIC you are running.

The coordination of all technical programming tasks as well as the design of all routines was done by Frank Ostrowski, one of the best assembler programmers in the world. He is also fully responsible for the UNIX version of the GFA-BASIC.

Dirk van Assche took over the porting of already written code to the Windows 3.0 and OS/2 operating systems. In addition he was responsible for the development of 80386 routines for the MS-DOS operating system. Furthermore, he implemented the MAT commands.

Roland Schütz ported the program code to the MS-DOS operating system. In addition, he developed all commands and functions for EMS memory management. He also had a major part in the development of drivers for various graphic cards and the implementation of the graphic and user interface routines.

Michael Wiegand, a programmer in the DTP area, provided C code for all graphic routines. In addition he developed the C code for all user interface routines for the MS-DOS operating system. The individual programmers then only had to translate this C segments into relevant assembler code.

## Introduction

---

Dr. Wolfgang Buscher was responsible for the management and administration of program development and, together with Frank Ostrowski, he wrote the handbooks for various GFA-BASIC PC versions.

Astrid Theus deserves special thanks for the tireless work with formatting, proofreading and the design of this handbook, which contributed in large part to GFA-BASIC PC appearing on the shelves as soon as possible after the programming has been completed.

Many thanks also to our Canadian translator Donald P. Maple, without whom it would not have been possible to finish this on time.

The result of all this effort is a programming language which today has no equal. With GFA-BASIC PC you have a tool which can be used to control almost all possible aspects of more than 90% of all computers in the world. This tool is extraordinarily flexible. The MS-DOS version has some 500, the WINDOWS version about 800, the OS/2 version roughly 1000 and the UNIX version about 900 commands and functions. The fundamental idea behind the development of these versions lies in how we interpreted the BASIC acronym, Beginners All Purpose Symbolic Instruction Code, i.e. a programming tool which, with its intuitive syntax, opens up all areas of the computer, for all programmers, including beginners.

The GFA-BASIC programming language uses an intuitive syntax for the abundance of its commands and functions, which were conceived more than ten years ago by Bill Gates. In addition, they use the control structures similar to Pascal and C programming languages and when used mainly for operating system operations are also similar to assembler. However, you need not be afraid that your code will degenerate into spaghetti code. GFA-BASIC forces you into structured programming. Whether you're BASIC, C or Pascal programmer, in GFA-BASIC you'll always find the useful elements from "your" programming language, be they structures, register or pointer operations, subroutines or variable passing. GFA-BASIC has implemented most useful parts from all of these programming languages. If you, nevertheless, decide to write certain routines in C or assembler, this is also no problem. GFA-BASIC enables you to select the

relevant routines and, by using the GFA-BASIC compiler, lets you bind them with the GFA-BASIC command library.

Congratulations on your decision to join the group of people whose programs can be used on more than 50 million computers and wish you success with your work.

### GFA Systemtechnik

# Handbook Conventions

This handbook is the reference portion of the GFA-BASIC documentation. It lists the various GFA-BASIC commands and functions in alphabetical order, explains their actions and describes their syntax, parameter passing and optional abbreviations. In addition, the command or function is demonstrated in a simple example.

You'll notice that we have intentionally omitted the page numbers. This has two grounds:

1. The alphabetical order makes the page numbers redundant and in certain circumstances confusing. You'll find the desired entry by simply opening the handbook and then paging back or forward.
2. A special attribute of GFA is the attention we pay to our customers. Certain wishes and ideas come often from our end users, relating to changes to certain commands or functions, or result in the implementation of new features. In such a case, the change of page numbering can occur which defies their purpose. If the page numbers are simply left out, it's possible to replace a command or function description with a new one or insert a new command in alphabetical order.

The header of each command or function indicates if this is a command or a function. In addition, it's often indicated in which category this command or function belongs (for example graphic). Some of these categories (for example graphic commands, commands for graphic user interface, operators etc.) are explained in detail in the Programmer's Handbook. The difference between the commands and functions is particularly important.

The commands always perform an instruction while functions first perform an instruction and then return one or more values. For example, the BOX  $x_1, y_1, x_2, y_2$  command draws a rectangle of the screen, while the GETSIZE( $x_1, y_1, x_2, y_2$ ) function returns the amount of memory which is used by this rectangle. In general: the functions can always be used in an IF conditional statement, for example



```
IF GETSIZE(20,20,300,200) <= 32000
  GET 20,20,300,200,a$
ELSE
  ALERT 1,"Segment too big",1,"Return",d%
ENDIF
```

A command can never be used in an IF conditional statement.

Except for a few exceptions (for example WINDGET or WINDFIND) the function parameter list is always within round brackets.

If applicable, when describing the syntax of a command or function the parameter list is shown behind the command or function. The parameters in square brackets (for example (STR\$(x[,n,m])), are optional, i.e. they need not be given when invoking the instruction.

The command or function syntax always contains the description of the category of necessary parameters. They mean the following:

**avar:** arithmetic variable. The variable must always be a numeric variable of any type.

**Example:**     DEC i%

**aexp:** arithmetic expression. The parameter can be a variable, constant or the result of any complex calculation.

**Example:**     FOR i%=a% TO 3\*a%\*TRUNC(SQR(PI\*180))

**svar:** string variable. This is a string variable which must always have the "\$" postfix.

**Example:**     GET 10,10,200,200,a\$

## Introduction

---

**sexp:** string expression. This is a string expression of any complexity.

**Example:**      `LEN("Hallo"+a$+RIGHT$(b$,10)+MID$(c$,3,6))`

**ivar:** integer variable. This is a whole number variable.

**Example:**      `MOUSE mx%,my%,mk%`

**iexp:** integer expression. This is a whole number expression of any complexity.

**Example:**      `m%=ADD(MUL(3,c%),MOD(k%,4))`

**bexp:** logical expression

**Example:**      `IF a%>b% & !k!`

The command syntax is followed by the abbreviation which can be used in the GFA-BASIC editor. Functions have no abbreviation.

And finally, a short explanation of the effect produced by calling the command or function. This is then clarified with an example. If necessary, further remarks may follow. The description of a command or function always ends with the declaration of the used command or function.



### **++ Command**

**Action:** increments a numeric variable.

**Syntax:** `x++`  
`x: avar`

**Abbreviation:** -

**Explanation:** `x++` increments the value of `x` by 1.

**Example:**

```
x=2.7
x++
PRINT x           // prints 3.7
```

**Remarks:** Although `++` can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of `x++`

```
x=x+1
x:=x+1
x+=1
INC x
SUB x,-1 or
ADD x,1
```

can be used also.

When integer variables are used `++` doesn't test for overflow!

**See**

**Also:** `INC`, `DEC`, `ADD`, `SUB`, `MUL`, `DIV`, `--`, `+=`, `-=`, `*=`, `/=`

## **+ = Command**

**Action:** adds a numeric expression to a numeric variable.

**Syntax:**

```
x += y
x:  avar
y:  aexp
```

**Abbreviation:** -

**Explanation:** `x += y` adds the expression `y` to the value in variable `x`.

**Example:**

```
x=17
x += 5*5
PRINT x           // prints 42
```

**Remarks:** Although `+=` can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of `x += y`

```
x=x+y
x:=x+y or
Add x,y
```

can be used also.

When integer variables are used `+=` doesn't test for overflow!

**See**

**Also:** DEC, INC, ADD, SUB, MUL, DIV, ++, --, -=, \*=, /=

### -- Command

**Action:** decrements a numeric variable.

**Syntax:** `x--`  
`x: avar`

**Abbreviation:** -

**Explanation:** `x--` decrements the value of `x` by 1.

**Example:** `x=2.7`  
`x--`  
`PRINT x` // prints 1.7

**Remarks:** Although `--` can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of `x--`

```
x=x-1
x:= x-1
x -= 1
DEC x
SUB x,1
ADD x,-1
```

can be used also.

When integer variables are used `--` doesn't test for overflow!

**See**

**Also:** `INC`, `DEC`, `ADD`, `SUB`, `MUL`, `DIV`, `++`, `+=`, `-=`, `*=`, `/=`

## - = Command

**Action:** subtracts a numeric expression from a numeric variable.

**Syntax:**

```
x -= y
x:  avar
y:  aexp
```

**Abbreviation:** -

**Explanation:** x -= y subtracts the expression y from the value in variable x.

**Example:**

```
x=57
x-=3*5
PRINT x           // prints    42
```

**Remarks:** Although -= can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of x -= y

```
x=x-y
x:= x-y or
SUB x,y
```

can be used also.

When integer variables are used -= doesn't test for overflow!

**See**

**Also:** DEC, INC, ADD, SUB, MUL, DIV, ++, --, +=, \*=, /=

### **\*/...\*/ Command**

**Action:** program comments

**Syntax:** \*/

**Abbreviation:** -

**Explanation:** /\*...\*/ is used to insert a comment within a line. Everything after a /\* until the \*/ is interpreted as a comment.

**Example:**

```
SCREEN 3
FOR i%=1 /* start the loop counter */ TO 10
  REM FOR...NEXT loop
  // FOR...NEXT loop
  /* FOR...NEXT loop
  ' FOR...NEXT loop
  PRINT i%
NEXT i%
```

**Remarks:** -

**See**

**Also:** //, ', REM, /\*



## **\* = Command**

**Action:** multiplies a numeric variable with a numeric expression.

**Syntax:**

```
x * = y
x:  avar
y:  aexp
```

**Abbreviation:** -

**Explanation:** `x * = y` multiplies the value in variable `x` with the expression `y`.

**Example:**

```
x=6
x * = 9
PRINT x           // prints    54
```

**Remarks:** Although `* =` can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of `x * = y`

```
x=x*y
x := x*y or
MUL x,y
```

can be used also.

When integer variables are used `* =` doesn't test for overflow!

**See**

**Also:** DEC, INC, ADD, SUB, MUL, DIV, ++, --, +=, -=, /=

### >> Function

**Action:** shifts a bit pattern right.

**Syntax:** `m >> n`  
*m,n: iexp*

**Explanation:** `m >> n` shifts the bit pattern of a 32-bit integer expressions `m`, `n` places right and thereby changes the value in `m`.

**Example:** `PRINT BIN$(202,16) // prints 0000000011001010`  
`PRINT BIN$(202 >>4,16) // prints 0000000000001100`

**Remarks:** `SHR(m,n)` is synonymous with `m >> n` and can be used instead. As long as the result of the shift does not exceed the given width, `m >> n` is equivalent to a division of `m` by  $2^n$ .

**See**

**Also:** `SHL, SHR, ROL, ROR, <<`

## < < Function

**Action:** shifts a bit pattern left.

**Syntax:**  $m << n$   
 $m, n: iexp$

**Explanation:**  $m << n$  shifts the bit pattern of a 32-bit integer expression  $m$ ,  $n$  places left and thereby changes the value in  $m$ .

**Example:**

```
PRINT BIN$(202,16)      // prints 0000000011001010
PRINT BIN$(202 << 4,16) // prints 0000110010100000
```

**Remarks:** SHL( $m,n$ ) is synonymous with  $m << n$  and can be used instead.

As long as the result of the shift does not exceed the given width,  $m << n$  is equivalent to a multiplication of  $m$  with  $2^n$ .

**See**

**Also:** SHL, SHR, ROL, ROR, > >

### **^^ Logical operator**

**Action:** logical XOR of true/false status of two values

**Syntax:** `i ^^ j`  
*i,j: function arguments*

**Explanation:** `i ^^ j` tests if one - and only one - of the function arguments is logically true (either-or).

**Example:**

```
IF i ^^ j // tests if either i or
//      j is not 0.
ENDIF
```

```
IF i>5 ^^ h<>"Hello" // tests either if i is
//                  greater than 5 or if h$
//                  is not equal to "Hello".
ENDIF
```

**Remarks:**

**See**

**Also:** `&&, ||, !`

## **^ = Command**

**Action:** performs an exclusive bit-wise OR on two bit patterns, whereby the first pattern must be in an integer variable.

**Syntax:**         $i \wedge = j$   
                  *i*:    *ivar*  
                  *j*:    *iexp*

**Explanation:**     $i \wedge = j$  sets in the integer variable *i* only the bits which are set in *i* but are clear in *j* and vice versa.

**Example:**        PRINT BIN\$(3,4)            // prints        0011  
                  PRINT BIN\$(10,4)        // prints        1010  
                  i%=3  
                  i% ^ = 10  
                  PRINT BIN\$(i%,4)        // prints        1001

**Remarks:**         $i = i \text{ XOR } j$  is synonymous with  $i \wedge = j$  and can be used instead.

**See**

**Also:**             $\&=$ ,  $|=$ ,  $\%=$

### || Logical operator

**Action:** logical OR of a true/false status of two values

**Syntax:** `i || j`  
*i,j: function arguments*

**Explanation:** `i || j` tests if at least one of the function arguments is logically true (and-or).

**Example:**

```
IF i || j // tests if i and/or j are
//      different from 0
ENDIF

IF i>5 || h$<>"Hello" // tests if i is greater than
//                    5 and/or if h$ is not
//                    equal to "Hello".
ENDIF
```

**Remarks:** -

**See**

**Also:** `&&, ^ ^, !`

## | Function

**Action:** performs a logical bit-wise OR on two bit patterns.

**Syntax:** `i | j`  
`i,j: iexp`

**Explanation:** `i | j` sets only the bits which are set in at least one of the two operands `i` or `j`.

**Example:**

```
PRINT BIN$(3,4)      // prints    0011
PRINT BIN$(10,4)     // prints    1010
PRINT BIN$(3 | 10,4) // prints    1011
```

**Remarks:** OR is synonymous with `|` and can be used instead:

```
PRINT BIN$(3 OR 10,4) // prints    1011
```

**See**

**Also:** AND, OR, XOR, IMP, EQV, &, ~

### | = Command

**Action:** performs a logical bit-wise OR on two bit patterns, whereby the first pattern must be in an integer variable.

**Syntax:** `i | = j`  
*i:* `ivar`  
*j:* `iexp`

**Explanation:** `i | = j` sets in the integer variable `i`, only the bits which are set in either `i` or `j`.

**Example:**

```
PRINT BIN$(3,4) // prints 0011
PRINT BIN$(10,4) // prints 1010
i%=3
i% |= 10
PRINT BIN$(i%,4) // prints 1011
```

**Remarks:** `i = i | j` and `i = i OR j` are synonymous with `i | = j` and can be used instead.

**See**

**Also:** `&=`, `^=`, `%=`



## // Command

**Action:** program comments

**Syntax:** //

**Abbreviation:** -

**Explanation:** // (double slash) can be used either at the beginning or within a GFA-BASIC program line. It can be followed by any comment since no syntax check is performed beyond this point. All subsequent characters are not considered to be a part of a command, function or variable. Within a line, // can't follow the PRINT command or be on a DATA line.

**Example:**

```
SCREEN 3
FOR i%=1 TO 10      // FOR... NEXT loop
  REM FOR...NEXT loop
  // FOR...NEXT loop
  /* FOR...NEXT loop
  '  FOR...NEXT loop
  PRINT i%
NEXT i%
```

**Remarks:** /\*, ' or REM can also be used at the beginning of a line instead of //.

**See**

**Also:** ', REM, /\*

### /= Command

**Action:** divides a numeric expression into a numeric variable.

**Syntax:**

$x /= y$   
*x:* *avar*  
*y:* *aexp*

**Abbreviation:** -

**Explanation:**  $x /= y$  divides the expression  $y$  into the value in variable  $x$ .

**Example:**

```
x=126
x /= 2+1
PRINT x           // prints    42
```

**Remarks:** Although  $/=$  can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of  $x /= y$

```
x=x/y
x:=x/y or
DIV x,y
```

can be used also.

When integer variables are used  $/=$  doesn't test for overflow!

**See**

**Also:** DEC, INC, ADD, SUB, MUL, DIV, ++, --, +=, -=, \*=

## && Logical operator

**Action:** logical AND of a true/false status of two values

**Syntax:** `i && j`  
*i,j: function arguments*

**Explanation:** `i && j` tests if both function arguments are logically true.

**Example:**

```
IF i && j          // tests if i and j are not 0
ENDIF

IF i>5 && h$<>"Hello" // tests if i is greater than
//                    5 and if h$ is not equal
//                    to "Hello".
ENDIF
```

**Remarks:** -

**See**

**Also:** `||, ^ ^, !`

### & Function

**Action:** performs a logical bit-wise AND of two bit patterns.

**Syntax:** `i & j`  
`i,j: iexp`

**Explanation:** `i & j` sets in the result only the bits which are set in both `i` and `j`.

**Example:**

```
PRINT BIN$(3,4)      // prints    0011
PRINT BIN$(10,4)     // prints    1010
PRINT BIN$(3 & 10,4) // prints    0010
```

**Remarks:** AND is synonymous with `&` and can be used instead:

```
PRINT BIN$(3 AND 10,4) // prints    0010
```

**See**

**Also:** AND, OR, XOR, IMP, EQV, |, ~

## & = Command

**Action:** a logical bit-wise AND of two bit patterns, whereby the first pattern must be in an integer variable.

**Syntax:**

```
i &= j  
i:  ivar  
j:  iexp
```

**Explanation:** `i &= j` sets, in the integer variable `i`, only the bits which are set in both `i` and `j`.

**Example:**

```
PRINT BIN$(3,4)      // prints    0011  
PRINT BIN$(10,4)     // prints    1010  
i%=3  
i% &= 10  
PRINT BIN$(i%,4)     // prints    0010
```

**Remarks:** `i = i & j` or `i = i AND j` are synonymous with `i &= j` and can be used instead.

**See**

**Also:** `^ =`, `| =`, `% =`

### % = Command

**Action:** calculates the modulus of an integer variable based on an integer expression and returns the result in the integer variable.

**Syntax:**

```
i %= j
i:  ivar
j:  iexp
```

**Explanation:**  $i \% = j$  calculates the modulus of integer variable  $i$  based on the integer expression  $j$  and writes the result in  $i$ .

**Example:**

```
i%=42
i% %= 6
PRINT i%           // returns    0
```

**Remarks:**  $i = i \text{ MOD } j$  is synonymous with  $i \% = j$  can be used instead.

**See**

**Also:**  $\& = , ^ = , | =$

## **{}** Function

**Action:** reads a double word (32 bits) from an address.

**Syntax:** {addr}  
*addr: address*

**Explanation:** Reads a double word (32 bits) from an address.

**Example:**

```

a=1.2345
PRINT HEX$({*a},8)'HEX$({*a+4},8)
PRINT
FOR i%=0 TO 7
  PRINT HEX$(PEEK(*a+i%),2)
NEXT i%
PRINT
PRINT a

```

*Handwritten notes: "Seq" above the first PRINT line, "Off" above the second PRINT line, and "For Seq: Off" written vertically to the left of the FOR loop.*

```

// Prints first 126E978D 3FF3C083, which is the
// internal representation of a as a longword and
// then 8D 97 6E 12 83 C0 F3 3F, which is the
// internal representation of a read in as bytes.

```

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset or a 32 bit integer value is given, where the 16 high bits represent the segment and the low 16 bits the offset.

LONG{ } or LPEEK() are synonymous with { } and can be used instead.

**See**

**Also:**

```

BYTE{ }, CARD{ }, INT{ }, WORD{ }, LONG{ },
SINGLE{ }, DOUBLE{ }, SHORT{ }, USHORT{ },
UWORD{ }, LPEEK{ }

```

### **{ } = Command**

**Action:** writes a double word (32 bits) to an address.

**Syntax:** {addr} = m  
*addr: address*  
*m: iexp*

**Explanation:** Writes a double word (32 bits) to an address.

**Example:** { \*a% } = { \*a% } + 1 // a slow a%++

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset or a 32 bit integer value is given, where the 16 high bits represent the segment and the low 16 bits the offset.

LONG{ } = or LPOKE are synonymous with { } = and can be used instead.

addr: see the { } function.

**See**

**Also:** LPOKE, BYTE{ } =, WORD{ } =, CARD{ } =, INT{ } =, LONG{ } =, SINGLE{ } =, DOUBLE{ } =, SHORT{ } =, USHORT{ } =, UWORD{ } =



## ~ Function

**Action:** a bitwise NOT

**Syntax:**        ~ i  
                  i:    iexp

**Explanation:**    ~ i inverts the bit pattern in i.

**Example:**

```
PRINT BIN$(3,32)
PRINT BIN$(10,32)
PRINT BIN$(~ 3,32)
PRINT BIN$(~ 10,32)

// prints
//
// 0000000000000000000000000000000011
// 00000000000000000000000000000001010
// 1111111111111111111111111111111100
// 1111111111111111111111111111110101
```

**Remarks:** NOT is synonymous with ~ and can be used instead.  
However, ~ has higher priority so

a% = ~b%+4 = (NOT b%)+4

a% = ~(b%+4) = NOT b%+4

**See**

**Also:** AND, OR, XOR, NOT, IMP, EQV, &, |

### ~ Command

**Action:** voids a numeric expression.

**Syntax:** ~ a  
*a: iexp*

**Abbreviation:** -

**Explanation:** ~ causes a calculated value or an integer expression returned from a function not to be put on stack or in a register. This means that the value is indeed calculated but because of ~ it's immediately "forgotten".

**Example:**

```
SCR 16
FOR i%=0 TO _x-1
    PLOT i%,(SINC(i%)+1)*_y/2
    ~SIN(COS(TAN(LOG(2.3))))
NEXT i%
```

```
// Causes a delay due to the highly calculation
// intensive expression.
// The same delaying effect can better be achieved by
// using PAUSE 1.
// In most cases, ~ is used together with INTR(), for
// example INTR($16,_AH=0) wait for a keypress with-
// out determining which key was actually pressed.
```

```
xo%=100
yo%=20
a$="POP-UP MENU |"+CHR$(255)+" L _1 | L _2| L _3"
~ POPUP(a$,xo%,yo%,1)
```

**Remarks:**

$\sim x$  is equivalent to  $\text{dummy}\% = x$

**VOID**  $x$  is equivalent to  $\text{dummy} = x$

**See****Also:**

**VOID**

### @ Command

**Action:** unconditional branch

**Syntax:** @procedurename[(parameterlist)] or  
@functionname[(parameterlist)]

*procedurename:* name of a subroutine declared  
with **PROCEDURE**

*functionname:* name of a function declared with  
**FUNCTION** or **DEFFN**

**Abbreviation:** -

**Explanation:** The @procedure- or @functionname causes an unconditional branch to the **PROCEDURE** or **FUNCTION** named in the command.

If the procedure or function was not declared correctly (for example invalid name or parameter list), an error is reported. **GOSUB.procedurename** or even just **procedurename** can be used instead of @procedurename.

**Example:**

```
Do
  INPUT "Enter text ";a$
EXIT IF a$=""
  @center_text(a$) // PROCEDURE-call
LOOP
END
PROCEDURE center_text(a$)
  LOCAL l%,x_pos%
  l%=LEN(a$)
  x_pos%=SHR(80-l%,1)
  LOCATE 5,x_pos%
  PRINT a$
RETURN
```

```
// In this example, the procedure 'center-text' is
// invoked in an endless loop.
```

```
DATA 12,3.4,5,6.7,8,9,12,14
DIM a(7)
sq=0
FOR i%=0 TO 7
  READ a(i%)
  ADD sq,a(i%)^2
NEXT i%
value=(sq-8*@am(7,a()))/7
PRINT "The standard deviation is: ";value
END
FUNCTION am(n% VAR vector())
  LOCAL i%,s
  IF n%<0
    RETURN error_code
  ELSE IF n%=0
    RETURN vector(0)
  ENDIF
  FOR i%=0 TO n%
    ADD s,vector(i%)
  NEXT i%
  RETURN DIV(s,SUCC(n%))
ENDFUNC
```

```
// In this example a function is called from the
// "value=..." line, and the value returned from the
// function is used as a part of an arithmetic
// expression.
```

```
DEFN nth_root(x,n%)=x^1/n%
PRINT "3rd root of PI = ";@nth_root(PI,3)
```

```
// In this example @ is used to jump to a one-line
// nth_root function.
```

## Commands and functions

---

**Remarks:** -

**See**

**Also:** GOSUB, ON...GOSUB, ON...event...GOSUB, FN,  
GOTO

## ! Logical Negation

**Action:** logical negation of a Boolean value

**Syntax:**       ! i  
              i:    *function argument*

**Explanation:**   ! i returns 0 when i is not zero and -1 when i equals 0.

**Example:**

```
i=32
PRINT ! i           // prints    0
i=0
PRINT ! i           // prints   -1
```

**Remarks:**       -

**See**

**Also:**           &&, ||, ^ ^

### ' Command

**Action:** program comments

**Syntax:** '

**Abbreviation:** -

**Explanation:** ' can be used either at the beginning or within a GFA-BASIC program line. If ' is at the start of GFA-BASIC line it can be followed by any comment since no syntax check is performed beyond this point. All subsequent characters are not considered a part of a command, function or variable. Within a line, ' can't follow the PRINT command or be on a DATA line.

**Example:**

```
SCREEN 3
FOR i%=1 TO 10 ' FOR ... NEXT loop
  REM FOR...NEXT loop
  // FOR...NEXT loop
  /* FOR...NEXT loop
  ' FOR...NEXT loop
  PRINT i%
NEXT i%
```

**Remarks:** // (double slash), /\* or REM can be used at the beginning of the line instead of '.

**See Also:** //, REM, /\*



## **AH Register variable**

**Action:** reads or writes the high byte of the AX register.

**Syntax:**        `x    =_AH or`  
                  `_AH = x`  
                  `x:    ixp`

**Explanation:** The industry standard PC uses eight general purpose registers, four segment registers, one program counter and one flag register.

The general purpose registers are composed of:

The accumulator register	AX
The base register	BX
The count register	CX
The data register	DX
The destination index register	DI
The source index register	SI
The stack pointer register	SP and
The base pointer register	BP

The segment registers are composed of:

The data segment register	DS
The extra segment register	ES
The code segment register	CS and
The stack segment register	SS

The program counter register is normally called IP and the flag register FL.

By placing an underdash in front of the abbreviation, these registers can be used within GFA-BASIC as variables, i.e. their values can be read and written. This is normally done when calling system interrupts with the GFA-BASIC command `INTR()`.

## Commands and functions

---

In case of general purpose registers AX, BX, CX and DX, the low and high byte of the registers can be referred to by using the GFA-BASIC variables `_AL`, `_AH`, `_BL`, `_BH`, `_CL`, `_CH`, `_DL` and `_DH`.

### Example:

```
DO
  ~INTR($16, _AX=0)
  IF _AL
    PRINT "ASCII code: "; _AL "Scan code: "; _AH
  ELSE
    PRINT "extended key code: "; _AH
  ENDIF
LOOP UNTIL _AL=27

// Reads a character from the keyboard buffer and
// returns, either the ASCII and Scan codes, or the
// extended key code. The loop is terminated by
// pressing the Esc key.
```

### Remarks:

-

### See

### Also:

`_AX`, `_AL`, `_BX`, `_BL`, `_BH`, `_CX`, `_CL`, `_CH`, `_DX`,  
`_DL`, `_DH`, `_DI`, `_SI`, `_SP`, `_BP`, `_DS`, `_ES`, `_CS`, `_SS`,  
`_IP`, `_FL`

## **\_AL Register variable**

**Action:** reads or writes the low byte of the AX register.

**Syntax:**

```
x = AL or
_AL = x
x: iexp
```

**Explanation:** see \_AH Register variable.

**Example:**

```
DO
  ~INTR($16, _AX=0)
  IF _AL
    PRINT "ASCII code: "; _AL "Scan code: "; _AH
  ELSE
    PRINT "extended key code: "; _AH
  ENDIF
LOOP UNTIL _AL=27

// Reads a character from the keyboard buffer and
// returns, either the ASCII and Scan codes, or the
// extended key code. The loop is terminated by
// pressing the Es key.
```

**Remarks:** -

**See**

**Also:** \_AX, \_AH, \_BX, \_BL, \_BH, \_CX, \_CL, \_CH, \_DX,  
\_DL, \_DH, \_DI, \_SI, \_SP, \_BP, \_DS, \_ES, \_CS, \_SS,  
\_IP, \_FL

### \_AX Register variable

**Action:** reads or writes the AX register.

**Syntax:**

```
x  = _AX or
_AX = x
x:  iexp
```

**Explanation:** see \_AH Register variable.

**Example:**

```
DO
  ~INTR($16,_AX=0)
  IF _AL
    PRINT "ASCII code: " ; _AL "Scan code: " ; _AH
  ELSE
    PRINT "extended key code: " ; _AH
  ENDIF
LOOP UNTIL _AL=27

// Reads a character from the keyboard buffer and
// returns, either the ASCII and Scan codes, or the
// extended key code. The loop is terminated by
// pressing the Esc key.
```

**Remarks:** -

**See**

**Also:** \_AL, \_AH, \_BX, \_BL, \_BH, \_CX, \_CL, \_CH, \_DX,  
\_DL, \_DH, \_DI, \_SI, \_SP, \_BP, \_DS, \_ES, \_CS, \_SS,  
\_IP, \_FL

## **BH Register variable**

**Action:** reads or writes the high byte of the BX register.

**Syntax:** `x = _BH or`  
`_BH = x`  
`x: iexp`

**Explanation:** see AH Register variable.

**Example:** `~INTR($10, _AH=$0B, _BH=0, _BL=$0A)`

```
// Sets the clipping rectangle and the background
// colour for graphic or text mode. In this example
// _BL=$0A sets the colour to light green.
```

**Remarks:** -

**See**

**Also:** `_AX, _AL, _AH, _BX, _BL, _CX, _CL, _CH, _DX,`  
`_DL, _DH, _DI, _SI, _SP, _BP, _DS, _ES, _CS, _SS,`  
`_IP, _FL`

### **\_BL Register variable**

**Action:** reads or writes the low byte of the BX register.

**Syntax:**

```
x  = _BL or
   _BL = x
x:  iexp
```

**Explanation:** see \_AH Register variable.

**Example:** `~INTR($10, _AH=$0B, _BH=0, _BL=$0A)`

```
// Sets the clipping rectangle and the background
// colour for graphic or text mode. In this example
// _BL=$0A sets the colour to light green.
```

**Remarks:** -

**See**

**Also:** \_AX, \_AL, \_AH, \_BX, \_BH, \_CX, \_CL, \_CH, \_DX,  
\_DL, \_DH, \_DI, \_SI, \_SP, \_BP, \_DS, \_ES, \_CS, \_SS,  
\_IP, \_FL

## **\_BP Register variable**

**Action:** reads or writes the BP register.

**Syntax:**

```
x  = _BP or
_BP = x
x:  iexp
```

**Explanation:** see \_AH Register variable.

**Example:**

```
SCREEN 16
~INTR($10,_AH=$11,_AL=$30,_BH=%01000011)
PRINT "The height of current character matrix: ";_CX
PRINT "Number of columns per line -1  : ";_DL
PRINT "The segment address of requested pointer :
";_ES
PRINT "The segment address of requested pointer :
";_BP

// Gets the address of the alternative character ROM
// 8*16, the contents of the interrupt vector $1F and
// the contents of the interrupt vector $43 and
// returns their values.
```

**Remarks:** -

**See**

**Also:** \_AX, \_AL, \_AH, \_BX, \_BL, \_BH, \_CX, \_CL, \_CH,  
\_DX, \_DL, \_DH, \_DI, \_SI, \_SP, \_DS, \_ES, \_CS, \_SS,  
\_IP, \_FL

### **\_BX Register variable**

**Action:** reads or writes the BX register.

**Syntax:**

```
x  = _BX or
_BX = x
x:  iexp
```

**Explanation:** see \_AH Register variable.

**Example:**

```
DO
~INTR($10, _AH=$04)
IF _AH
PRINT "The line with light characters in text
mode: " _DH
PRINT "The column with light characters in text
mode: " _DL
PRINT "The line with light characters in graphic
mode: " _CH
PRINT "The column with light characters in
graphic mode: " _BX
ELSE
PRINT "Position of light characters cannot be
determined"
ENDIF
LOOP UNTIL LEN(INKEY$)

// Checks if the light characters are on and returns
// their position on the screen.
// The loop is terminated by pressing the Esc key.
```

**Remarks:** -

**See**

**Also:** \_AX, \_AL, \_AH, \_BL, \_BH, \_CX, \_CL, \_CH, \_DX,  
\_DL, \_DH, \_DI, \_SI, \_SP, \_BP, \_DS, \_ES, \_CS, \_SS,  
\_IP, \_FL



## **C Variable**

**Action:** contains the number of possible colours for current SCREEN.

**Syntax:** C

**Explanation:** By using X, Y and C the programs can be written regardless of the resolution or window size.

C returns the value 16 for EGA and VGA modes, 4 for CGA, 2 for HGC, 256 for MCGA and 0 for text modes.

**Example:**

```
SCREEN 16
PRINT C           // prints 16
```

**Remarks:** -

**See**  
**Also:** X, Y

### **\_CH Register variable**

**Action:** reads or writes the high byte of the CX register.

**Syntax:**

`x` = `_CH` or  
`_CH` = `x`  
`x`: *iexp*

**Explanation:** see `_AH` Register variable.

**Example:**

```
~INTR($1A,_AH=$02)
IF IBTST(_FL,0)
    PRINT "Clock: Hours:Minutes:Seconds = ";
    PRINT _CH;";";_CL;";";_DH
ELSE
    PRINT "Flat clock battery or no clock"
ENDIF
REPEAT
UNTIL LEN(INKEY$)

// Reads the time from the battery backed up real
// time clock.
```

**Remarks:** -

**See**

**Also:** `_AX`, `_AL`, `_AH`, `_BX`, `_BL`, `_BH`, `_CX`, `_CL`, `_DX`,  
`_DL`, `_DH`, `_DI`, `_SI`, `_SP`, `_BP`, `_DS`, `_ES`, `_CS`, `_SS`,  
`_IP`, `_FL`

## **\_CL Register variable**

**Action:** reads or writes the low byte of the CX register.

**Syntax:**

```
x  = _CL or
_CL = x
x:  iexp
```

**Explanation:** see \_AH Register variable.

**Remarks:**

```
~INTR($1A, AH=$02)
IF !BTST(_FL,0)
  PRINT "Clock: Hours:Minutes:Seconds = ";
  PRINT _CH;":";_CL;":";_DH
ELSE
  PRINT "Flat clock battery or no clock"
ENDIF
REPEAT
UNTIL LEN(INKEY$)

// Reads the time from the battery backed up real
// time clock.
```

**Remarks:** -

**See**

**Also:** \_AX, \_AL, \_AH, \_BX, \_BL, \_BH, \_CX, \_CH, \_DX,  
\_DL, \_DH, \_DI, \_SI, \_SP, \_BP, \_DS, \_ES, \_CS, \_SS,  
\_IP, \_FL

### CX Register variable

**Action:** reads or writes the CX register.

**Syntax:**

```
x = _CX or
_CX = x
x: iexp
```

**Explanation:** see AH Register variable.

**Example:** `~INTR($15,_AH=$86,_CX=2000,_DX=0)`  
`// Pauses for 2 milliseconds.`

**Remarks:** -

**See**

**Also:** AX, AL, AH, BX, BL, BH, CL, CH, DX,  
DL, DH, DI, SI, SP, BP, DS, ES, CS, SS,  
IP, FL

## **\_DATA Function and Command**

**Action:** positions the data pointer for READ at a value previously determined with the \_DATA function.

**Syntax:** \_DATA = x  
x: *avar*

**Abbreviation:** -

**Explanation:** \_DATA - like RESTORE - influences the data pointer. While RESTORE always positions the data pointer to the first value on a DATA line, \_DATA enables also for positioning of the data pointer to a particular place within a DATA line. This place must be defined beforehand using the \_DATA function.

**Example:**

```
DIM dp%(100)
DATA 1,2,3,4,5,6,7,8,9
DATA 13,24,328,3242,1,0
i%=0
WHILE _DATA
  dp%(i%)=_DATA           // The current position of
  i%++                     // data pointer is determined
  READ a                   // by using the _DATA
//                           function
WEND                       // and saved in
i%--                       // dp%(i%).
//
FOR j%=i% DOWNT0 0
  _DATA=dp%(j%)           // By using the
  READ a                   // _DATA command, the data
//                           pointer
  PRINT a'                 // is set to a position
//                           previously returned from
```

## Commands and functions

---

```
NEXT j%           // the _DATA function and
EDIT             // the corresponding value
//              // from the DATA line is read
//              // in.

// prints
// 0 1 3242 328 24 13 9 8 7 6 5 4 3 2 1
```

**Remarks:**

-

**See**

**Also:**

RESTORE, \_DATA

## **DH Register variable**

**Action:** reads or writes the high byte of the DX register.

**Syntax:** `x| = _DH` or  
`_DH = x|`  
`x: iexp`

**Explanation:** see AH Register variable.

**Example:**

```
~INTR($1A, _AH=$02)
IF IBTST(_FL,0)
  PRINT "Clock: Hours:Minutes:Seconds = ";
  PRINT _CH;":":_CL;":":_DH
ELSE
  PRINT "Flat clock battery or no clock"
ENDIF
REPEAT
UNTIL LEN(INKEY$)

// Reads the time from the battery backed up real
// time clock.
```

**Remarks:** -

**See**

**Also:** AX, AL, AH, BX, BL, BH, CX, CL, CH,  
DX, DL, DI, SI, SP, BP, DS, ES, CS, SS,  
IP, FL

### **\_DI Register variable**

**Action:** reads or writes the high byte of the DI register.

**Syntax:**

```
x  = _DI or
_DI = x
x:  iexp
```

**Explanation:** see \_AH Register variable.

**Example:**

```
DATA $E3,$04,$AB,$40,$E2,$FC,$CB,-1
mc$=""
DO
    READ a%
    EXIT IF a%<0
    mc$=mc$+chr$(a%)
LOOP
dim a$(1000)
CALL (v:mc$)(_ES=SWAP(V:a$(0)),_DI=0,_CX=1001,_AX=0)
FOR i%=0 TO 1000
    PRINT a$(i%)
NEXT i%
```



```
// assembler source
//
// .MODEL LARGE
// .code
//          entry  proc far
//          jcxz   ret0
// lop0:    stosw
//          inc    ax
//          loop   lop0
// ret0:    ret          ;retf
//          entry  endp
// end
```

**Remarks:** -

**See**

**Also:**

\_AX, \_AL, \_AH, \_BX, \_BL, \_BH, \_CX, \_CL, \_CH,  
\_DX, \_DL, \_DH, \_SI, \_SP, \_BP, \_DS, \_ES, \_CS, \_SS,  
\_IP, \_FL

### **\_DL Variable**

**Action:** reads or writes the low byte of the DX register.

**Syntax:**

```
x  = _DL or
_DL = x
x:  iexp,ivar
```

**Explanation:** see \_AH Register variable.

**Example:**

```
PROCEDURE stdout(s$)
  LOCAL i%
  FOR i%=0 TO LEN(s$)-1
    ~INTR($21,_AH=2,_DL=BYTE{V:a$+i%})
  NEXT i%
RETURN

// Prints a string through DOS (using output re-
// direction).
```

**Remarks:** -

**See**

**Also:** \_AX, \_AH, \_AL, \_BX, \_BL, \_BH, \_CX, \_CL, \_CH,  
\_DX, \_DH, \_DI, \_SI, \_SP, \_BP, \_DS, \_ES, \_CS, \_SS,  
\_IP, \_FL

## \_DX Register variable

**Action:** reads or writes the CX register.

**Syntax:**

```
x = _DX or
_DX = x
x: iexp
```

**Explanation:** see \_AH Register variable.

**Example:**

```
i%=64
DO
  i%++
  ~INTR($14,_AH=$01,_DX=$00,_AL=i%)
  IF _AH
    PRINT "Sending character ";CHR$(i%);"will be
        transferred"
  ELSE
    PRINT "Transmission error!"
  ENDF
LOOP UNTIL i%=90

// Sends all capital letters through the serial port
// (COM 1).
```

**Remarks:** -

**See**

**Also:** \_AX, \_AL, \_AH, \_BX, \_BL, \_BH, \_CX, \_CL, \_CH,  
\_DL, \_DH, \_DI, \_SI, \_SP, \_BP, \_DS, \_ES, \_CS, \_SS,  
\_IP, \_FL

### **\_EAX, \_EBX, \_ECX, \_EDX, \_ESI, \_EDI, \_EBP, \_ESP, \_EFL Variables**

**Action:** access to 386 registers for INTR or CALL

**Syntax:** x = \_EAX or  
\_EAX = x  
x: *iexp,ivar*

**Explanation:** The 80386 general purpose registers (also 80386DX, 80386SX and 80486) are not only 16, but 32 bits wide. The normal 16 (and 8) bit registers, AX etc. can still be used as before. The DOS interrupt (INTR(\$21)) as well as most other interrupts use only the lower half of these registers. In the 386 version of GFA-BASIC these extended registers are passed in full.

**Example:** -

**Remarks:** -

**See**

**Also:** \_AX, \_BX, \_CX, \_DX, \_DI, \_SI, \_SP, \_BP

## **\_FL Variable**

**Action:** access to the flag register after INTR or CALL

**Syntax:** `x = _FL`  
`x: ivar`

**Explanation:** In addition to normal registers, the 8086 processor also contains a flag register, used to return errors for example.

The bits of these registers are called:

----ODIT SZ-A-P-C

which is to say Overflow, Direction, Interrupt, Trace, Sign, Zero, Auxiliary, Parity and Carry.

They show the processor status after INTR or CALL. DOS (INTR(\$21) and other calls normally set the carry flag in case of error, so that

IF ODD(\_FL)

reacts to an error.

**Example:**

```
a$="@@NIX@@.NIX"+CHR$(0) // name of a non existent
// subdirectory.
_DX = CARD(V:a$) // address of the name to
// DS:DX
_DS = SWAP(V:a$)
~INTR($21,_AH=$3A)
IF ODD(_FL)
    PRINT "Error deleting a subdirectory"
    PRINT "Error ";_AX
ENDIF
```

# Commands and functions

Remarks:

See

Also: `_AX ...`

## **PSP Variable**

**Action:** A variable containing the address of the program segment prefix (PSP) for the GFA-BASIC interpreter.

**Syntax:** `m = _PSP`  
`m: iexp`

**Explanation:** The program segment prefix (PSP) is a 256 byte long block of memory with the following layout:

OFFSET	Length	Description
\$00	2 bytes	interrupt \$20 call (terminate program)
\$02	2 bytes	end address of assigned memory block (segment address only)
\$041	1 byte	reserved
\$05	5 bytes	FAR CALL for DOS function distributor (interrupt \$21)
\$0A	4 bytes	interrupt \$22 address of the termination handle
\$0E	4 bytes	interrupt \$23 address of Control Break handle
\$12	4 bytes	interrupt \$24 address of the critical error handle
\$16	2 bytes	PSP of the parent process
\$18	20 bytes	handle table
\$2C	2 bytes	segment address of the environment block
\$2E	4 bytes	reserved
\$32	2 bytes	handle table size
\$34	4 bytes	address of handle table (if not \$12)
\$38	24 bytes	reserved

## Commands and functions

---

\$50	2 bytes	interrupt \$21 DOS call
\$52	1 byte	RET FAR
\$53	9 bytes	reserved
\$5C	16 bytes	closed standard File Control Block #1
\$6C	20 bytes	closed standard File Control Block #2
\$80	1 byte	number of characters on the command line excluding the CR and followed by the start of the standard Disk Transfer Address (DTA)
\$81	127 bytes	command line which starts with space and ends with a CR.

### Example:

```
a%=_PSP+$2C}  
DO  
  a$=CHAR{a%}  
  EXIT IF LEN(a$)=0  
  PRINT a$  
  a%+=SUCC(LEN(a$))  
LOOP  
  
// Returns the complete environment.
```

### Remarks:

BASEPAGE is synonymous with \_PSP and can be used instead.

### See

### Also:

BASEPAGE, FSETDTA, FGETDTA, FSFIRST, FSNEXT



## **SI Register variable**

**Action:** reads or writes the SI register.

**Syntax:**        `x = _SI` or  
                 `_SI = x`  
`x:`        *exp*

**Explanation:** see `_AH` Register variable.

**Example:**        `CLS`  
                 `A$=SPACE$(1000)`  
                 `PRINT "Drive ";`  
                 `REPEAT`  
                     `b$=UPPER$(INKEY$)`  
                 `UNTIL b$>="A" AND b$<="Z"`  
                 `_DL =ASC(b$) & 31`                `//`  
                 `_DS =SWAP(V:A$)`  
                 `_SI =V:A$`  
                 `_INTR($21,_AH=$47)`  
                 `IF EVEN(_FL)`  
                     `PRINT "Directory : ";b$;"\";CHAR{V:A$}`  
                 `ELSE`  
                     `PRINT "Error ";_AX`  
                 `ENDIF`

`// Draws a box and erases the mouse pointer if it`  
`// happens to be within this box. When the mouse`  
`// leaves the box it appears again.`

**Remarks:**        -

**See**

**Also:**            `_AX, _AL, _AH, _BX, _BL, _BH, _CX, _CL, _CH,`  
                 `_DX, _DL, _DH, _DI, _SP, _BP, _DS, _ES, _CS, _SS,`  
                 `_IP, _FL`

### **\_X Variable**

**Action:** contains the width of the current window or screen.

**Syntax:** \_X

**Explanation:** By using \_X, \_Y and \_C the programs can be written regardless of the resolution or window size. When no window is open or after a WIN #0, \_X returns the width of the whole screen. Otherwise, \_X returns the width of the current window (OPENW or WIN).

**Example:**

```
SCREEN 16
PRINT _X'_Y           // prints 640 350
```

**Remarks:** -

**See**

**Also:** \_Y, \_C

## **\_Y Variable**

**Action:** contains the height of the current window or screen.

**Syntax:** \_Y

**Explanation:** By using \_X, \_Y and \_C the programs can be written regardless of the resolution or window size.

When no window is open or after a WIN #0, \_Y returns the height of the whole screen. Otherwise, \_Y returns the height of the current window (OPENW or WIN).

**Example:**

```
SCREEN 16
PRINT _X'_Y           // prints 640 350
```

**Remarks:** -

**See**

**Also:** \_X, \_C

### ABS() Numeric function

**Action:** returns the absolute value of a numeric expression.

**Syntax:** `ABS(x)`  
*x: aexp*

**Explanation:** -

**Example:**

<code>PRINT ABS(-210)</code>	<code>// prints</code>	<code>210</code>
<code>PRINT ABS(5-10)</code>	<code>// prints</code>	<code>5</code>
<code>PRINT ABS(0)</code>	<code>// prints</code>	<code>0</code>

**Remarks:** The returned value from `ABS()` depends on the sign of the `x` argument:

for  $x < 0$  returns  $-x$ ,  
for  $x = 0$  returns 0 and  
for  $x > 0$  returns  $x$ .

**See**

**Also:** `SGN`

## ACOS() Trigonometrical function

**Action:** returns the arc cosine of a numeric expression.

**Syntax:** ACOS(*x*)  
*x*: *aexp*

**Explanation:** ACOS(*x*) expects as function argument *x* the quotient between hypotenuse and the side forming the angle (in a right-angled triangle) and returns the angle in radians. It follows, therefore, that the value of *x* ranges between -1 (equivalent to COS(PI)) and 1 (equivalent to COS(0)).

**Example:**

```
PRINT ACOS(-1)           // prints 3.14...
PRINT ACOS(0)            // prints 1.57...
PRINT ACOS(1)            // prints 0
PRINT ACOS(COS(PI))      // prints 3.14...
```

**Remarks:** ACOS() is the reverse function of COS().

**See**

**Also:** SIN(), SINQ(), COS(), COSQ(), TAN(), ASIN(),  
ATN(), ATAN()

### ADD Command

**Action:** adds a numeric expression to a numeric variable.

**Syntax:**        `ADD x,y`  
                  `x:    avar`  
                  `y:    aexp`

**Abbreviation:**    -

**Explanation:**    `ADD x,y` adds the expression `y` to the value in variable `x`.

**Example:**        `x=17`  
                  `ADD x,5*5`  
                  `PRINT x`        `// prints    42`

**Remarks:**        Although `ADD` can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of `ADD x,y`

`x=x+y`  
`x:=x+y` or  
`x+=y`

can be used also.

When integer variables are used `ADD` doesn't test for overflow!

**See**

**Also:**            `DEC, INC, SUB, MUL, DIV, ++, --, +=, -=, *=, /=`

## ADD() Function

**Action:** adds up two or more integer expressions.

**Syntax:**        `ADD(i,j[,m,...])`  
                 *i,j,m,...:        iexp*

**Explanation:**    `ADD(i,j[,m,...])` calculates the sum of integer expressions *i, j, m,...*

**Example:**

```
PRINT ADD(2*4,4+3)        // prints        15
PRINT ADD(10,17,5,10)    // prints        42
1%=ADD(2*4,3+4)
PRINT 1%                    // prints        15
```

**Remarks:**        The `ADD()`, `SUB()`, `MUL()` and `DIV()` functions can be mixed freely with each other. For example:

```
1%=ADD(5^3,4*20-3) or
1%=ADD(5^3,SUB(MUL(4,20),3))
```

**See**

**Also:**              `SUB()`, `MUL()`, `DIV()`, `MOD()`, `PRED()`, `SUCC()`

# ALERT Command

**Action:** draws an Alert Box on the screen.

**Syntax:** ALERT *i*,*a*\$,*j*,*b*\$,*k*  
*i*,*j*: *iexp*  
*k*: *ivar*  
*a*\$,*b*\$: *sexp*

**Abbreviation:** ale *i*,*a*\$,*j*,*b*\$,*k*

**Explanation:** An Alert Box is a special form of a Dialog Box. It is used when a point in a program is reached where the program is to be cancelled, a certain branch is to be taken or some other user decision is to be made.

The first integer expression, *i*, determines which symbol will be included in the Alert Box together with the message. The following symbols are available:

- i* symbol
- 0 no symbol
- 1 exclamation mark
- 2 question mark
- 3 stop sign

*a*\$ contains the message which is to be displayed in the Alert Box. If the text is too long for one line it can be split in up to 4 lines by using "|".

*b*\$ contains up to five possible alternatives for user response.

*j* indicates which of these alternatives is the default. This alternative is then selected by pressing the Return key. The alternatives are numbered from 1 to 5 and are separated from each other by a "|".

*k* contains the number of the alternative which was actually selected.



**Example:**

```
i%=2
a$="Which procedure should|be executed next"
j%=1
b$="Input | Calculate | Print | File output | CANCEL"
k%=0
ALERT i%,a$,j%,b$,k%

// Creates an Alert Box with a question mark as
// symbol and the message: "Which procedure should be
// executed next". The default alternative is
// "Input". The alternatives are:
// Input, Calculate, Print, File output and CANCEL.
// k% contains the number of the selected alter-
// native.
// ALERT 3,"An error|has occurred",1,"CANCEL",k%
// creates an Alert Box with a stop sign as symbol
// and CANCEL as the only alternative.
```

**Remarks:**

In addition to the menu bar and pop-up menus, the Alert Box is a third possible way of communication between the program and the user.

**See****Also:**

MENU, POPUP

### AND Function

**Action:** performs a logical bit-wise AND of two bit patterns

**Syntax:** `i AND j`  
`i,j: iexp`

**Explanation:** `i AND j` sets in the result only the bits which are set in both `i` and `j`.

**Example:**

```
PRINT BIN$(3,4)      // prints    0011
PRINT BIN$(10,4)     // prints    1010
PRINT BIN$(3 AND 10,4) // prints    0010
```

**Remarks:** `&` is synonymous with `AND` and can be used instead:

```
PRINT BIN$(3 & 10,4) // prints    0010
```

**See**

**Also:** `OR`, `XOR`, `IMP`, `EQV`, `&`, `|`, `~`

## AND() Function

**Action:** performs a logical bit-wise AND of two bit patterns

**Syntax:**           AND(*i,j*)  
                  *i,j: iexp*

**Explanation:**   AND(*i,j*) sets in the result only the bits which are set in both *i* and *j*.

**Example:**

```
PRINT BIN$(3,4)           // prints 0011
PRINT BIN$(10,4)          // prints 1010
PRINT BIN$(AND(3,10),4)   // prints 0010
```

**Remarks:**       -

**See**

**Also:**           OR(), XOR(), IMP(), EQV()

### ARCOSH() Transcendental function

**Action:** returns the hyperbolic cosine area of a numeric expression.

**Syntax:**       ARCOSH(*x*)  
*x*:    *aexp*

**Explanation:**   The hyperbolic cosine area applies to all real numbers greater than or equal to 1. It is obtained with the function:

$$\text{ARCOSH}(x) = \text{LOG}(x + \text{SQR}(x^2 - 1))$$

The function  $y = \text{ARCOSH}(x)$  returns in *y* a real number greater than or equal to 0.

**Example:**       PRINT ARCOSH(2.14)       // prints 1.39425...  
                  PRINT ARCOSH(COSH(2.14)) // prints 2.14

**Remarks:**       ARCOSH() is the reverse function of COSH().

**See**

**Also:**            SINH(), COSH(), TANH(), ARSINH(), ARTANH()

## ARRAYFILL Command

**Action:** initializes a numerical array of any type with a value.

**Syntax:**        `ARRAYFILL a()=x`  
                 *a(): any numeric or Boolean array*  
                 *x:    aexp*

**Abbreviation:**    `arr a(),x`

**Explanation:**    The `ARRAYFILL a(),x` command can be used on all numeric and Boolean arrays with up to 6 dimensions. The complete array `a()`, including all dimensions, is filled with the expression `x`. If `x` is a floating point expression, a `TRUNC(x)` is performed if the arrays to be filled are integer arrays. By default, all dimensioned numeric arrays are cleared with 0, while all Boolean arrays are initialized with `FALSE` - which is also 0. When an `ARRAYFILL` is performed on a Boolean array with a value other than 0 (`FALSE`), it always results in initialization with -1 (`TRUE`).

**Example:**        `DIM a%(10),b%(3,4),c%(2,3,4),d|(1,2,3,4),e!(5)`  
                 `ARRAYFILL a%(),17.4`  
                 `ARRAYFILL b%(),13`  
                 `ARRAYFILL c%(),17`  
                 `ARRAYFILL d|(),9`  
                 `ARRAYFILL e!(),TRUE`  
                 `//`  
                 `PRINT a(1)                // prints     17.4`  
                 `PRINT b%(1,1)            // prints     13`  
                 `PRINT c%(1,2,2)          // prints     17`  
                 `PRINT d|(1,1,2,2)        // prints     9`  
                 `PRINT e!(3)               // prints    -1`

## Commands and functions

---

Remarks:

-

See

Also:

MAT SET

## Arrays

**DIM, DIM?**

**OPTION BASE, MAT BASE**

**ARRAYFILL, MAT SET**

**ERASE**

**OPTION BASE 0**

**OPTION BASE 1**

**MAT BASE 0**

**MAT BASE 1**

OPTION BASE 0 and OPTION BASE 1 determine if the indexing of array elements should start with 0 or with 1 respectively. The default is OPTION BASE 0. MAT BASE 0 and MAT BASE 1 commands are only relevant with OPTION BASE 0. MAT BASE 1 makes the indexing of elements in one and two dimensional arrays of double variables start with element 1 and 1,1 respectively, in spite of already selected OPTION BASE 0. MAT BASE 0 sets the element indexing after MAT BASE 1 back to 0.

The commands MAT BASE 0 and MAT BASE 1 have no effects after OPTION BASE 1; since the indexing after this option always start from 1.

**Example:**

```
OPTION BASE 0
DIM a(3)
FOR i%=0 TO 3
  a(i%)=i%
NEXT i%
MAT PRINT a()
MAT BASE 1
MAT PRINT a()
MAT BASE 0
MAT PRINT a()
```

A one-dimensional array for 4 double variables is defined first. This is followed by writing the values from 0 to 3 into this array. Printing the array results in 0,1,2,3. After MAT BASE 1, however, 1,2,3 is displayed, but after MAT BASE 0 the printing results in 0,1,2,3 again.

**NOTE!** OPTION BASE works with arrays of all variable types and in any dimension. MAT BASE works only with one and two dimensional, double variable arrays. MAT BASE 1 from OPTION BASE 0 does not reduce the memory occupied by an array. It only starts the indexing from 1 instead of 0.

### **ARRAYFILL x(),y MAT SET a(),y**

**x:** name of an array composed of any numeric variables (Double, Word,...)  
**a:** name of an array composed of double variables  
**y:** aexp

**ARRAYFILL:** Sets all elements in a dimensioned, numeric array to a given value.

**Example:**

```
DIM a(5,2,3), a%(10), a!(2,3)
ARRAYFILL a(),3.7      // sets all elements in a()
                        // to value 3.7
                        //
ARRAYFILL a%( ),114     // sets all elements in a%( )
                        // to value 114
                        //
ARRAYFILL a!( ),TRUE    // sets all elements in a!( )
                        // to value -1
```



**MAT SET:** relevant only for one and two dimensional arrays of double variables where it works exactly like ARRAY-FILL.

**Example:**

```
DIM a(4,4)
MAT SET a(),PI      // sets all elements in a()
//                  to the value of PI =
//                  3.14...
```

**ERASE:** Deletes all specified arrays. If several arrays are to be deleted together they should be listed after the command ERASE. and separated by commas:

**Example:**

```
DIM a(1,2,5), a!(16,2), a$(10)
.
.
ERASE a(),a!(),a$()
```

**NOTE!:** MAT CLR a() performs an ARRAYFILL a(),0 - it does not delete the array a().

### ARRPTR() Function

**Function:** returns the address of a variable of any type.

**Syntax:**        **ARRPTR(x)**  
                 *x:    variable or field name*

**Abbreviation:**    \*

**Explanation:**    **ARRPTR(a())** and **ARRPTR(a\$)** return the addresses of array and string descriptors respectively.

**Example:**

```
PRINT ARRPTR(a())        // prints the address of a()
//
PRINT *a$                // prints the address of
//                        a$ descriptor
PRINT ARRPTR(n%)        // prints the address of n%
PRINT *n%                // prints the address of n%
```

**Remarks:**        \*x is synonymous with **ARRPTR()** and can be used instead.

**See**

**Also:**             **VARPTR(), V:, PEEK(), DPEEK(), LPEEK()**

## ARSINH() Transcendental function

**Action:** returns the hyperbolic sine area of a numeric expression.

**Syntax:** ARSINH(*x*)  
*x*: *aexp*

**Explanation:** The hyperbolic sine area is obtained with the function:  
$$\text{ARSINH}(x) = \text{LOG}(x + \text{SQR}(x^2 + 1))$$

**Example:**  

```
PRINT ARSINH(2.14)           // prints 1.50454...  
PRINT ARSINH(SINH(2.14))    // prints 2.14
```

**Remarks:** ARSINH() is the reverse function of SINH().

**See**

**Also:** SINH(), COSH(), TANH(), ARCOSH(),  
ARTANH()

### ARTANH() Transcendental function

**Action:** returns the hyperbolic tangent area of a numeric expression.

**Syntax:** ARTANH(*x*)  
*x*: *aexp*

**Explanation:** The hyperbolic tangent area is obtained with the function:

$$\text{ARTANH}(x) = \text{LOG}((1+x)/(1-x))/2$$

**Example:**

```
PRINT ARTANH(-0.5)           // prints -0.54930...
PRINT ARTANH(TANH(-0.5))    // prints -0.5
```

**Remarks:** ARTANH() is the reverse function of TANH(). The hyperbolic cotangent area is obtained with:

$$\text{ARCOTH}(x) = 1/\text{ARTANH}(x)$$

**See**

**Also:** SINH(), COSH(), TANH(), ARSINH(), ARCOSH()

## ASC() Function

**Action:** determines the ASCII value of the first character in a string.

**Syntax:** ASC(a\$)  
*a\$: sexp*

**Abbreviation:** -

**Explanation:** ASC(a\$) returns the ASCII code of the first character in a\$. If a\$ is blank a 0 is returned.

**Example:** PRINT ASC("TEST") // prints 84 since 84 is the  
// ASCII code for T.

**Remarks:** -

**See**

**Also:** CVI(), CVL(), CVS(), CVD()

### ASIN() Trigonometrical function

**Action:** returns the arc sine of a numeric expression.

**Syntax:** ASIN(x)  
*x:* *anexp*

**Explanation:** ASIN(x) expects as function argument x the quotient between the hypotenuse and the side opposite the angle (in a right-angled triangle) and returns the angle in radians. It follows from this that the value of "x" ranges between -1 (equivalent to SIN(-PI/2)) and 1 (equivalent to SIN(PI/2)).

**Example:**

```
PRINT ASIN(-1)      // prints    -1.57...
PRINT ASIN(1)       // prints    1.57...
PRINT ASIN(SIN(PI)) // prints    0
```

**Remarks:** ASIN() is the reverse function of SIN() in the range  $[-\pi/2, \pi/2]$ .

**See**

**Also:** SIN(), SINQ(), COS(), COSQ(), TAN(), ACOS(), ATN(), ATAN()

## ATAN() Trigonometrical function

**Action:** returns the arc tangent of a numeric expression.

**Syntax:** ATAN(*x*)  
*x*: *anexp*

**Explanation:** ATAN(*x*) expects as function argument *x* the quotient between the two short sides in a right-angled triangle and returns the angle in radians.

**Example:**

```
PRINT ATAN(-PI)      // prints    -1.26...
PRINT ATAN(1)        // prints    0.78...
PRINT ATAN(PI/4)     // prints    0.66...
PRINT ATAN(TAN(PI/4)) // prints    0.78...
```

**Remarks:** ATAN() is the reverse function of TAN().  
ATN() is synonymous with ATAN() and can be used instead.

**See**

**Also:** SIN(), SINQ(), COS(), COSQ(), TAN(), ASIN(),  
ACOS(), ATN()

## ATN() Trigonometrical function

**Action:** returns the arc tangent of a numeric expression.

**Syntax:** ATN(*x*)  
*x*: *anexp*

**Explanation:** ATN(*x*) expects as function argument *x* the quotient between the two short sides in a right-angled triangle and returns the angle in radians.

**Example:**

```
PRINT ATN(-PI)      // prints  -1.26...
PRINT ATN(1)        // prints   0.78...
PRINT ATN(PI/4)     // prints   0.66...
PRINT ATN(TAN(PI/4)) // prints   0.78...
```

**Remarks:** ATN() is the reverse function of ATAN(). ATN() is synonymous with ATAN() and can be used instead.

**See**

**Also:** SIN(), SINQ(), COS(), COSQ(), TAN(), ASIN(), ACOS(), ATAN()



## A.10 Mathematical Functions

These functions are available in the GFA-BASIC PC environment.

**Function**  
**Description**

**ABS** (X) : Returns the absolute value of X.  
**ACOS** (X) : Returns the arccosine of X, in radians.  
**ASIN** (X) : Returns the arcsine of X, in radians.  
**ATN** (X) : Returns the arctangent of X, in radians.

**EXP** (X) : Returns the value of e raised to the power of X.  
**LOG** (X) : Returns the natural logarithm of X.  
**LOG10** (X) : Returns the base-10 logarithm of X.  
**POW** (X, Y) : Returns the value of X raised to the power of Y.

**SIN** (X) : Returns the sine of X, in radians.  
**COS** (X) : Returns the cosine of X, in radians.  
**TAN** (X) : Returns the tangent of X, in radians.

**ATN2** (Y, X) : Returns the arctangent of Y/X, in radians.  
**SGN** (X) : Returns the sign of X, -1 for negative, 1 for positive, and 0 for zero.

### BASEPAGE Variable

**Action:** the variable containing the address of the basepage (i.e. the program segment prefix; PSP) of the GFA-BASIC interpreters.

**Syntax:** `m = BASEPAGE`  
`m:` *iexp*

**Explanation:** The program segment prefix (PSP) is a 256 byte long block of memory with the following layout:

OFFSET	Length	Description
\$00	2 bytes	interrupt \$20 call (terminate program)
\$02	2 bytes	end address of assigned memory block (segment address only)
\$041	1 byte	reserved
\$05	5 bytes	FAR CALL for DOS function distributor (interrupt \$21)
\$0A	4 bytes	interrupt \$22 address of the termination handle
\$0E	4 bytes	interrupt \$23 address of Control Break handle
\$12	4 bytes	interrupt \$24 address of the critical error handle
\$16	2 bytes	PSP of the parent process
\$18	20 bytes	handle table
S2C	2 bytes	segment address of the environment block
\$2E	4 bytes	reserved
\$32	2 bytes	handle table size

\$34	4 bytes	address of handle table (if not \$12)
\$38	24 bytes	reserved
\$50	2 bytes	interrupt \$21 DOS call
\$52	1 byte	RET FAR
\$53	9 bytes	reserved
\$5C	16 bytes	closed standard File Control Block #1
\$6C	20 bytes	closed standard File Control Block #2
\$80	1 byte	number of characters on the command line excluding the CR and followed by the start of the standard Disk Transfer Address (DTA)
\$81	127 bytes	command line which starts with space and ends with a CR.

**Example:**       PRINT CHAR{BASEPAGE+129}

                  // Displays the command line.

**Remarks:**     \_PSP is synonymous with BASEPAGE and can be  
                  used instead.

**See**

**Also:**         \_PSP, FSETDTA, FGETDTA, FSFIRST, FSNEXT

### BCHG() Function

**Action:** changes the status of a bit in an integer expression.

**Syntax:** `BCHG(m,n)`  
*m,n: iexp*

**Explanation:** `BCHG(m,n)` sets the n-th bit in the integer expression m to 1 (if this bit is 0), or to 0 (if this bit is 1).

**Example:**

```
PRINT BIN$(BCHG(10,0),4) // prints 1011
i%=BCHG(10,0)
PRINT i%                  // prints 11
```

**Remarks:** -

**See**

**Also:** `BCLR()`, `BSET()`, `BTST()`

## BCLR() Function

**Action:** clears one bit in an integer expression.

**Syntax:** BCLR(*m*,*n*)  
*m*,*n*: *iexp*

**Explanation:** BCLR(*m*,*n*) clears the *n*-th bit in the integer expression *m* (the bit is set to 0).

**Example:**

```
PRINT BIN$(BCLR(10,3),4) // prints 0010
i%=BCLR(10,3)
PRINT i%                  // prints 2
```

**Remarks:** -

**See**

**Also:** BSET(), BTST(), BCHG()

### BGET Command

**Action:** fast read of files saved with BPUT.

**Syntax:** BGET #n,addr,count  
*n,addr,count: iexp*

**Abbreviation:** bge #n,addr,count

**Explanation:** BGET (block get) is used to read files stored with BPUT (block put). A channel for the file must be opened first with OPEN "i",#n,name\$. addr contains the address where in memory the file should be loaded. count defines how much data should BGET read from the file.

**Example:**

```
dim a%(999)
addr%=V:a%(0)
count%=(V:a(1)-V:a(0))*200
OPEN "i",#1,"A:\TEST.DAT"
BGET "A:\TEST.DAT",addr%,count%
CLOSE #1#

// Reads the first 200 values from file TEST.DAT
// starting from the address V:a(0) into the array
// a%().
```

**Remarks:** BGET and BPUT can also be used to save and read parts of a file.

**See Also:** BSAVE, BLOAD, BPUT

## BIN\$() Function

**Action:** converts an integer expression to a binary string representation.

**Syntax:** BIN\$(m[,n])  
*m,n: iexp*

**Abbreviation:** -

**Explanation:** After conversion the binary representation of integer expression *m* is returned as a plain string.

The parameter *n* is optional and determines how many places (1 to 33) should be used to represent the number. If *n* is greater than the number of places needed to represent *m* the converted number is padded with leading zeros.

**Example:**

```
PRINT BIN$(17)           // prints 10001
PRINT BIN$(25,6)         // prints 011001
```

**Remarks:** -

**See**

**Also:** OCT\$(), HEX\$(), DEC\$()

### BLOAD Command

**Action:** fast load of files saved with BSAVE.

**Syntax:** BLOAD a\$[,addr]  
*a\$: sexp; file name*  
*addr: iexp*

**Abbreviation:** blo a\$[,addr]

**Explanation:** BLOAD (block load) is used to read the file a\$ previously stored with BSAVE (block save). The parameter addr is optional and it contains the address where in memory the file should be loaded. If addr is left out, the last address used BSAVE is used again.

**Example:**  
dim a%(999)  
addr%=V:a%(0)  
BSAVE "A:\TEST.DAT",addr%

**Remarks:** BSAVE and BLOAD always access the whole file.

**See**

**Also:** BSAVE, BPUT, BGET



## BMOVE Command

**Action:** copies an area of memory.

**Syntax:** BMOVE from\_addr%, to\_addr%, i%  
*from\_addr%, to\_addr%: address*  
*i%: iexp*

**Abbreviation:** bmo f\_addr%,t\_addr%,i%

**Explanation:** BMOVE is used to copy memory areas. The copy is performed from address from\_addr% to the address to\_addr%. The number of bytes to copy is specified in i%.

**Example:**

```
DIM a%(20,3), b%(20,3)
FOR i%=0 TO 20
  FOR j%=1 TO 3
    a%(i%,j%)=RANDOM(2000-1000)
  NEXT j%
NEXT i%
BMOVE V:a%(0,0),V:b%(0,0),DIM?(a%)*4
```

```
// Two integer arrays are dimensioned first.
// Following that, a%() is filled with random
// numbers.
// V:a%(0,0) returns the address of the first element
// in a%() and V:b%(0,0) the address of the first
// element in b%(). LONG (%) integer variables
// require four bytes of memory. The number of
// elements in a%() is determined by DIM?(a%()).
// DIM?(a%()*4 then results in the number of bytes
// copied.
```

## Commands and functions

---

**Remarks:** The copying of array a%() into array b%() in the above example can also be done with

```
FOR i%=0 TO 20
  FOR j%=0 TO 3
    b%(i%,j%)=a%(i%,j%)
  NEXT j%
NEXT i%
```

The BMOVE command, however, requires less memory and is - depending on the contents being copied - up to 100 times faster.

**See**  
**Also:**

-

## BOUND() Function

**Action:** bounds test

**Syntax:** BOUND(*n,lo,hi*)  
*n,lo,hi:* *iexp*

**Explanation:** The BOUND(*n,lo,hi*) function tests whether the parameter *n* lies within bounds of *lo* and *hi* (inclusive). This means that when *n* < *lo* or *n* > *hi* an error message is reported. Otherwise *n* is returned unchanged.

**Example:**

```
DIM a%(49)
FOR i%=1 TO 20
  q%=RAND(49)+1
  WHILE a%(q%)
    q%++
  WEND
  INC a%(q%)
NEXT i%
```

```
// This programs selects 20 random numbers between 1
// and 49 without repetition. The frequency of the
// number (zero or once) is noted in array a%().
// If RAND() returns a number for the second time the
// next higher number is taken instead.
// After ten test runs an error (array index too big)
// appears six times.
// To locate this error the line q%++ can, for
// example, be changed to
//
// q%=BOUND(q%+1,1,49)
//
// This will cause an error (Bound Error) on the line
// where q% is modified (q%++). In this way the place
// where the range is exceeded is easier to find.
```

## Commands and functions

---

**Remarks:** The BOUND() function serves to find program errors by early discovery of any range violations.

**See**  
**Also:** BOUNDB(), BOUNDW(), BOUNDC()

## BOUNDB() Function

**Action:** bounds test

**Syntax:** BOUNDB(*n*)  
*n*: *iexp*

**Explanation:** The BOUNDB(*n*) function tests if the parameter *n* fits in a BYTE. This means that when  $n < 0$  or  $n > 255$  an error message is reported. Otherwise *n* is returned unchanged.

**Example:** `i|=BOUNDB(a|*b|)`

```
// Tests if the result of the multiplication of byte
// variables a| and b| is within the allowed range
// for a byte variable. If it isn't an error message
// appears.
```

**Remarks:** The BOUNDB() function serves to find program errors by early discovery of any range violations.

**See**

**Also:** BOUNDC(), BOUNDW(), BOUND()

### BOUNDC() Function

**Action:** bounds test

**Syntax:** BOUNDC(*n*)  
*n*: *iexp*

**Explanation:** The BOUNDC(*n*) function tests if the parameter *n* fits in an unsigned word (CARD). This means that when  $n < 0$  or  $n > 65535$  an error message is reported. Otherwise *n* is returned unchanged.

**Example:** DPOKE addr%,BOUNDC(b%)

```
// Tests if b% can be written to the address addr%  
// with a DPOKE, without truncating the value in b%.  
// If not, an error message appears.
```

**Remarks:** The BOUNDC() function serves to find program errors by early discovery of any range violations.

**See**

**Also:** BOUNDB(), BOUNDW(), BOUND()

## BOUNDW() Function

**Action:** bounds test

**Syntax:** BOUNDW(*n*)  
*n*: *iexp*

**Explanation:** The BOUNDW(*n*) function tests if the parameter *n* fits in a word. This means that when  $n < -32768$  or  $n > 32767$  an error message is reported.

Otherwise *n* is returned unchanged.

**Example:** `i&=BOUNDW(a&*b&)`

```
// Tests if the result of the multiplication of
// variables a& and b& is within the range of a word
// variable. If not, an error appears.
```

**Remarks:** The BOUNDW() function serves to find program errors by early discovery of any range violations.

**See**

**Also:** BOUNDC(), BOUNDB(), BOUND()

### BOX Graphic command

**Action:** draws a rectangle.

**Syntax:** BOX x1,y1,x2,y2  
x1,y1,x2,y2: *exp*

**Abbreviation:** -

**Explanation:** BOX x1,y1,x2,y2 draws a rectangle with diagonal corner coordinates x1,y1 (upper left) and x2,y2 (lower right).

**Example:** CLS  
BOX 10,10,100,100

**Remarks:** -

**See**

**Also:** PBOX, RBOX, PRBOX



## BPUT Command

**Action:** fast save of an area of memory to disk.

**Syntax:** BPUT #*n*,*addr*,*count*  
*n*,*addr*,*count*: *iexp*

**Abbreviation:** bpu #*n*,*addr*,*count*

**Explanation:** An area of memory can be saved to disk (RAM disk, hard disk etc.) using BPUT (block put) and loaded back in with BGET (block get). The channel #*n* must be opened first with OPEN "o",#*n*,*names*\$. The integer expression *addr* contains the start address of the memory to be saved. In addition, *count* must specify the length of the file.

**Example:**

```
dim a%(999)
FOR i%=0 TO 999
  a%(i%)=RAND(1000)
NEXT i%
addr%=V:a%(0)
count%=(V:a%(1)-V:a%(0))*1000
OPEN "o",#1,"A:\TEST.DAT"
BPUT #1,addr%,count%
CLOSE #1
```

**Remarks:** The saving of memory with BPUT is similar to BSAVE. In contrast to BSAVE, BPUT saves the data through a previously opened channel under a previously defined file name.

**See**

**Also:** BLOAD, BSAVE, BGET

### BSAVE Command

**Action:** fast save of an area of memory to disk.

**Syntax:** BSAVE a\$,addr,count  
*a\$:* *sexp; file name*  
*addr,count:* *iexp*

**Abbreviation:** bsa a\$,addr,count

**Explanation:** An area of memory can be saved to disk (RAM disk, hard disk etc.) using BSAVE (block save) and loaded back in with BLOAD (block load). The integer expression *addr* contains the start address of the memory to be saved. In addition, *count* must specify the length of the file *a\$*.

**Example:**

```
dim a%(999)
FOR i%=0 TO 999
    a%(i%)=RAND(1000)
NEXT i%
addr%=V:a%(0)
count%=(V:a%(1)-V:a%(0))*1000
BSAVE "A:\TEST.DAT",addr%,count%
```

**Remarks:** The saving of files using BSAVE is - depending on the medium - 5 to 10 times faster than with OPEN...PRINT# ...CLOSE. Even the memory needed by BSAVE is - depending on the file - up to three times smaller. BSAVE and BLOAD always access the whole file.

**See**

**Also:** BLOAD, BPUT, BGET

## BSET() Function

**Action:** sets one bit in an integer expression.

**Syntax:** BSET(*m,n*)  
*m,n: iexp*

**Explanation:** BSET(*m,n*) sets the *n*-th bit in the integer expression *m* (the bit is set to 1).

**Example:** PRINT BIN\$(BSET(10,2),4) // prints 1110  
i%=BSET(10,2)  
PRINT i% // prints 14

**Remarks:** -

**See**

**Also:** BCLR(), BTST(), BCHG()

### BTST() Function

**Action:** tests the bit status in an integer expression.

**Syntax:** BTST(*m,n*)  
*m,n: iexp*

**Explanation:** BTST(*m,n*) returns -1.(true) when the *n*-th bit in the integer expression *m* is set, and 0 (false) if it's not.

**Example:** PRINT BTST(10,3) // prints -1

**Remarks:** -

**See**

**Also:** BCLR(), BSET(), BCHG()

## BYTE() Function

**Action:** masks with 255

**Syntax:** BYTE(*m*)  
*m*: *iexp*

**Explanation:** BYTE performs an AND between *m* and 255 (\$FF).

**Example:**

```
s%=16
m%=235
a=1586
PRINT BIN$(m%,s%)           // 0000000011101011
PRINT BIN$(BYTE(m%),s%)     // 0000000011101011
PRINT STRING$(s%,"-")       // -----
PRINT BIN$(a,s%)            // 0000011000110010
PRINT BIN$(BYTE(a),s%)      // 0000000000110010
PRINT BIN$(a AND 255,s%)    // 0000000000110010
PRINT BIN$(a MOD 256,s%)    // 0000000000110010
PRINT BYTE(a)               // 50
```

**Remarks:** -

**See**

**Also:** WORD(), CARD(), SHORT(), USHORT(),  
UWORD()

### BYTE{} Function

**Action:** reads a byte from an address.

**Syntax:** `BYTE{addr}`  
*addr: address*

**Abbreviation:** -

**Explanation:** An address is composed of a segment and an offset. To specify an address a 16 bit integer expression is given to represent the segment, followed by a colon and a 16 bit integer expression for the offset.

**Example:**

```
PRINT BYTE{*a()} // prints the first byte of
// a() descriptor
```

**Remarks:** addr: see {} function.

**See**

**Also:** `PEEK()`, `WORD{}`, `CARD{}`, `INT{}`, `LONG{}`, `{}`,  
`SINGLE{}`, `DOUBLE{}`, `SHORT{}`, `USHORT{}`,  
`UWORD{}`

## BYTE{} = Command

**Action:** writes a byte-sized (8 bits) signed integer expression to an address.

**Syntax:** BYTE{addr} = m  
*addr: address*  
*m: a byte-sized signed integer expression*

**Abbreviation:** -

**Explanation:** An address is composed of a segment and an offset. To specify an address a 16 bit integer expression is given to represent the segment, followed by a colon and a 16 bit integer expression for the offset.

**Example:** FOR I=1 TO 0160 STEP 2  
BYTE{\_TS:I%}=\$F1  
NEXT I%

// Changes the text attributes of the first line  
// without actually changing any characters.

**Remarks:** addr: see {} function.

**See**

**Also:** POKE(), CARD{}=, WORD{}=, INT{}=,  
LONG{}=, {}=, SINGLE{}=, DOUBLE{}=,  
SHORT{}=, USHORT{}=, UWORD{}=

### C: Function

**Action:** invokes a subroutine written in C or assembler.

**Syntax:** C:addr%([x,y...])

**Explanation:** The C: function calls a C or an assembler subroutine at address addr%. The parameters x,y,... are passed in brackets. Parameter passing is the same as in C, where 16-bit values are the default. However, by using L:xx, longword parameters can also be passed.

C: routines are invoked with a "far call". The parameters are saved on stack in reversed order.

C: routines should not change the SI, DI, BP and SP registers, as well as DS and SS. The return value is stored in AX (low word) as well as DX (high word).

**Example:**

```
DATA $55,$8B,$EC,$57,$C4,$7E,$06,$33,  
DATA $C0,$8B,$4E,$0A,$E3,$04,$AB,$40,  
DATA $E2,$FC,$5F,$5D,$CB,-1  
mc$=""  
DO  
  READ a$  
  EXIT IF a%<0  
  mc$=mc$+CHR$(a$)  
//
```



```
FOR i%=0 TO 1000
  PRINT a&(i%)'
NEXT i%
```

```
// assembler source
//
//          .MODEL LARGE
//          .code
// entry    proc    far
//          push    bp      ; save bp
//          mov     bpsp
//          push    di      ; save di
//          les     di,[bp+6] ; first parameter, far ptr
//          xor     ax,ax    ; z to 0
//          mov     cx[bp+10] ; counter to cx
//          jcxz    ret0
// lop0:    stosw
//          inc     ax
//          loop    lop0
// ret0:    pop     di      ; restore registers
//          pop     bp
//          ret     ; retf
// entry    endp
//          end
```

**Remarks:**

-

**See**

**Also:**

CALL

### CALL Command

**Action:** invokes a subroutine written in C or assembler.

**Syntax:** CALL addr%[parameters]

**Explanation:** The CALL function calls a C or an assembler subroutine at address addr%.

Optionally, addr can be followed with register assignments similar to INTR(). The return value is received back in \_AX etc.

When the address is not specified in an % variable it must be given in brackets.

**Example:**

```
DATA $E3,$04,$AB,$40,$E2,$FC,$CB,-1
mc$=""
DO
    READ a%
    EXIT IF a%<0
    mc$=mc$+chr$(a%)
LOOP
'
dim a$(1000)
'
CALL (v:mc$)(_ES=SWAP(V:a$(0)),_DI=0,_CX=1001,_AX=0)
'
FOR i%=0 TO 1000
    PRINT a$(i%)
NEXT i%
```

```
// assembler source
//
//          .MODEL LARGE
//          .code
// entry  proc      far
//          jcxz     ret0
// lop0:   stosw
//          inc      ax
//          loop     lop0
// ret0:   ret              ;retf
// entry  endp
//          end
```

**Remarks:** -

**See**

**Also:** C:

## CARD() Function

**Action:** performs an AND 65535

**Syntax:** CARD(*m*)  
*m*: *iexp*

**Explanation:** Limits *m* to 16 bits by clearing the bits 16 to 31.

**Example:**

```
s%=32
m%=2096
a=100000
PRINT BIN$(m%,s%)
PRINT BIN$(CARD(m%),s%)
PRINT STRING$(s%,"-")
PRINT BIN$(a,s%)
PRINT BIN$(CARD(a),s%)
PRINT BIN$(a AND 65535,s%)
PRINT BIN$(a MOD 65536,s%)
PRINT CARD(a)

// prints
//
// 00000000000000000000100000110000
// 00000000000000000000100000110000
// -----
// 000000000000000011000011010100000
// 00000000000000001000011010100000
// 00000000000000001000011010100000
// 00000000000000001000011010100000
// 34464
```

**Remarks:** USHORT() and UWORD() are synonymous with CARD() and can be used instead.

**See**

**Also:** BYTE(), WORD(), SHORT(), USHORT(), UWORD()

### CARD{} Function

**Action:** reads a word (16 bits) from an address.

**Syntax:** CARD{addr}  
*addr: address*

**Abbreviation:** -

**Explanation:** An address is composed of a segment and an offset. To specify an address a 16 bit integer expression is given to represent the segment, followed by a colon and a 16 bit integer expression for the offset.

**Example:**

```
PRINT CARD{*a$}      // prints the first word of
//                  a$ descriptor

PRINT CARD{$40:$1E}  // prints the first word at
//                  offset $1E in segment $40
```

**Remarks:** USHORT{} and UWORD{} are synonymous with CARD{} and can be used instead.

addr: see {} function.

**See**

**Also:** BYTE{}, WORD{}, INT{}, LONG{}, {}, SINGLE{}, DOUBLE{}, SHORT{}, USHORT{}, UWORD{}

## CARD{} = Command

**Action:** writes a word (16 bits) to an address.

**Syntax:** CARD{addr} = m  
*addr: address*  
*m: iexp*

**Abbreviation:** -

**Explanation:** An address is composed of a segment and an offset. To specify an address a 16 bit integer expression is given to represent the segment, followed by a colon and a 16 bit integer expression for the offset.

**Example:** -

**Remarks:** addr: see {} function.

**See**

**Also:** BYTE{}, WORD{}, INT{}, LONG{}, {},  
SINGLE{}, DOUBLE{} SHORT{}, USHORT{},  
UWORD{}

## CASE Condition test

**Action:** a part of the SELECT...CASE structure. It contains the condition test.

**Syntax:** CASE condition

**Explanation:** see "SELECT...CASE Structure"

**Remarks:** -



## CEIL() Numeric function

**Action:** returns the smallest integer number greater than or equal to a numeric expression.

**Syntax:**            `CEIL(x)`  
                      *x: aexp*

**Explanation:**    `CEIL(x)` is equivalent to rounding up towards + .

**Example:**            `PRINT CEIL(3*1.2)`        `// prints`        `4`  
                      `PRINT CEIL(3*-1.2)`    `// prints`        `-3`

**Remarks:**            -

**See**

**Also:**                `FLOOR()`, `INT()`, `TRUNC()`, `FIX()`, `FRAC()`

### CFLOAT() Function

**Action:** converts an integer into a floating point number.

**Syntax:** CFLOAT(*i*)  
*i*: *exp*

**Explanation:** -

**Example:** a%=12345  
b=CFLOAT(a%)  
PRINT b // prints 12345

**Remarks:** -

**See**

**Also:** CINT()

## CHAIN Command

**Action:** loads a GFA-BASIC program in memory and runs it.

**Syntax:** CHAIN a\$  
*a\$: sexp*

**Abbreviation:** -

**Explanation:** The CHAIN command loads a GFA-BASIC program in computer memory and runs it. If the extension is left out .GFA is used.

**Example:** CHAIN "C:\GFA\TEST.GFA"

```
// Loads TEST.GFA from subdirectory GFA in partition  
// C and runs it.
```

**Remarks:** -

**See**

**Also:** RUN, EXEC

### CHAR{} Function

**Action:** reads a string from an address until the next zero byte.

**Syntax:** CHAR{addr}  
*addr: address*

**Abbreviation:** -

**Explanation:** The CHAR{addr} function requires an address as a parameter. CHAR{} reads the data from this address until the next zero byte and returns this as a string.

**Example:** a\$="Ford"+CHR\$(0)  
b\$="Prefect"+CHR\$(0)  
PRINT CHAR{V:a\$};"";CHAR{V:b\$}

// prints Ford Prefect

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset or a 32 bit integer value is given, where the 16 high bits represent the segment and the low 16 bits the offset.

addr: see the {} function.

**See**

**Also:** BYTE{}, WORD{}, INT{}, CARD{}, LONG{}, {}, SINGLE{}, DOUBLE{}, SHORT{}, USHORT{}, UWORD{}

## CHAR{} = Command

**Action:** writes a string to an address.

**Syntax:** CHAR{addr} = a\$  
*addr: address*  
*a\$: sexp*

**Abbreviation:** -

**Explanation:** The CHAR{addr}=a\$ command requires as parameters an address and a character string expression. CHAR{} writes the string a\$ from this address and terminates it with a zero.

**Example:**

```
b$="Ford Prefect"
b%=len(b$)
a$=SPACE$(b%+1)
CHAR{V:a$}=b$
PRINT a$'ASC(RIGHT$(a$))
// prints Ford Prefect 0
```

**Remarks:** The string format with a zero byte terminator is used in particular by C and operating system routines.

An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset or a 32 bit integer value is given, where the 16 high bits represent the segment and the low 16 bits the offset.

addr: see the {} function.

**See**

**Also:** BYTE{}, WORD{}, INT{}, CARD{}, LONG{}, {}, SINGLE{}, DOUBLE{}, SHORT{}, USHORT{}, UWORD{}

# CHDIR Command

**Action:** sets the current directory.

**Syntax:** CHDIR a\$  
a\$: *sexp; name of current directory*

**Explanation:** CHDIR sets the current directory. Since it is impossible to change the drive with CHDIR, this command always defaults to current drive. CHDIR must be followed by the path name. If a\$ contains only the backslash ("\"), the change to the root directory of the current drive is performed.

There are two special abbreviations for CHDIR: "." and "..". "." is an alternative way to define the current subdirectory and ".." for the parent directory. Let's assume that the current subdirectory contains the directory Test, which in turn contains directories A1 and A2. \Test\A1 is the current path. In this case CHDIR "..\A2" will change the current path to \Test\A2.

**Example:**

```
CHDRIVE 1           // drive A is the current
//                 drive
CHDIR "\Test"       // A:\Test
CHDIR "A1"          // A:\Test\A1
CHDIR "..\A2"       // A:\Test\A2
```

**Remarks:** -

**See**

**Also:** -

## CHDRIVE Command

**Action:** sets the current drive.

**Syntax:** CHDRIVE *n* or *n\$*

*n*: *iexp*

*n\$*: *sexp*

**Explanation:** CHDRIVE (change drive) sets the current drive. If an input or output command does not contain a drive, all inputs and outputs default to the current drive. *n* can assume the values from 1 to 16, and these values correspond to drives A to P. Instead of a drive number, CHDRIVE can also take a string whose first character is the drive letter.

**Example:**

```
CHDRIVE 1           // sets drive A as the
//                  current drive
```

**Remarks:** -

**See**

**Also:** -

### CHR\$() Function

**Action:** converts an integer expression into a character.

**Syntax:** CHR\$(m)  
*m: iexp*

**Explanation:** CHR\$(m) returns the ASCII character with the ASCII code m.

The value of m is a number between 0 and 255.

**Example:** PRINT CHR\$(65) // prints the character A  
// since 65 is its ASCII code

**Remarks:** -

**See**

**Also:** MKI\$(),MKL\$(), MKS\$(), MKD\$()



## CINT() Function

**Action:** converts a floating point number to an integer.

**Syntax:** CINT(a)  
*a:* *aexp*

**Explanation:** CINT (a) converts the floating point number a into a rounded integer value.

**Example:**

```
a=12345.789
b%=CINT(a)
PRINT b           // prints      12346
```

**Remarks:** -

**See**

**Also:** CFLOAT()

### CIRCLE Graphic command

**Action:** draws a circle.

**Syntax:** CIRCLE x,y,r[,w1,w2]  
x,y,r,w1,w2: *ixp*

**Abbreviation:** ci x,y,r[,w1,w2]

**Explanation:** CIRCLE x,y,r[,w1,w2] draws a circle with the radius r around the centre with the coordinates x,y. Additionally, the start (w1) and end (w2) angles can be given to define an arc. The angles w1 and w2 are given in whole degrees steps.

**Example:** CLS  
CIRCLE 100,100,20,90,180

// Draws an arc at 100,100 with the radius of 20.

**Remarks:** -

**See**

**Also:** PCIRCLE, ELLIPSE, PELLIPSE

## CLEAR Command

**Action:** deletes all variables and arrays.

**Syntax:** CLEAR

**Abbreviation:** -

**Explanation:** This command cannot be used inside loops or subroutines. A CLEAR is performed automatically when the program starts up.

**Example:**

```
x=2
y=2
CLEAR
PRINT x,y           // prints 0 0
```

**Remarks:** -

**See**

**Also:** CLR, ERASE

### CLEARW# Command

**Action:** deletes the contents of a window.

**Syntax:** CLEARW #n  
n: *ixp*

**Abbreviation:** -

**Explanation:** When graphic mode is on, CLEARW #n deletes the contents of the window previously opened with OPENW #n.

**Example:**

```
SCREEN 16                // EGA mode
OPENW #1,10,10,200,100,-1
REPEAT
    a$=inkey$
UNTIL a$=chr$(27)
FOR i%=1 TO 5
    PRINT "TEST TEST TEST"
NEXT i%
REPEAT
    a$=INKEY$
UNTIL a$=CHR$(27)
CLEARW #1
PRINT "Window contents will be deleted"
PRINT "and the same text will be rewritten"
REPEAT
    a$=INKEY$
UNTIL a$=CHR$(27)
CLOSEW #1
SCREEN 3
EDIT
```

```
// Draws a window on the screen and writes five
// identical lines with TEST TEST TEST. When the Esc
// key is pressed the window is cleared and the text
```

```
// "Window contents will be deleted..." is written in
// the window. When the Esc key is pressed again the
// window is closed, the test mode is turned on and
// the GFA-BASIC editor is invoked.
```

**Remarks:** -

**See**

**Also:** OPENW, TITLEW, INFOW, SIZEW, TOPW,  
FULLW, CLOSEW

### CLIP Graphic command

**Action:** sets the bounds for graphic output.

**Syntax:** CLIP *x,y,w,h* [ OFFSET *x0,y0*] or  
CLIP *x1,y1* TO *x2,y2* [ OFFSET *x0,y0*] or  
CLIP #*n* [ OFFSET *x0,y0*] or  
CLIP OFFSET *x0,y0* or  
CLIP OFF

*x,y,w,h,x0,x1,x2,y0,y1,y2,n: iexp*

**Abbreviation:** cli *x,y,w,h* [ OFFSET *x0,y0*] or  
cli *x1,y1* TO *x2,y2* [ OFFSET *x0,y0*] or  
cli #*n* [ OFFSET *x0,y0*] or  
cli offs *x0,y0* or  
cli of

**Explanation:** CLIP *x,y,w,h* [ OFFSET *x0,y0*] limits the graphic output to a defined rectangle. Using the optional OFFSET *x0,y0* command the origins of the graphic output can be set to the point with the x-coordinate *x0* and y-coordinate *y0*. *x* and *y* define the x- and y-coordinates for the upper left corner of the clipping window. *w* (width) and *h* (height) define the width and height of the clipping window relative to *x* and *y*.

CLIP *x1,y1* TO *x2,y2* defines the upper left corner of the clipping rectangle with *x1* and *y1*, and the lower right corner with *x2* and *y2*.

CLIP #*n* enables clipping of the output to a window with the number *n*.

CLIP OFFSET *x0,y0* defines the origins for the output to the point with coordinates *x0,y0*.

CLIP OFF turns the clipping off.

**Example:**

```
CLIP 10,10 100,100 // limits the graphic output
//                to a window with the
//                following coordinates:
//                10,10      upper left
//                110,10     upper right
//                10,110     lower left
//                110,110    lower right
//
CLIP 10,10 TO 100,100 // limits the graphic output
//                to a window with the
//                following coordinates:
//                10,10      upper left
//                100,10     upper right
//                10,100     lower left
//                100,100    lower right
//
OPEN W1           // opens the window #1.
//
CLIP #1           // sets clipping to the area
//                covered by window #1.
//
CLIP OFF          // turns the clipping off.
```

**Remarks:**

The clipping does not apply to the GET and PUT commands.

**See****Also:**

-

# CLOSE Command

**Action:** closes a channel.

**Syntax:** CLOSE [#n]  
*n:* *exp*

**Abbreviation:** cl [#n]

**Explanation:** CLOSE [#n] closes a channel to a file or peripheral device, previously opened with OPEN. The parameter *n* contains the number of the channel to close. If no channel number is given all open file channels are closed.

**Example:** see OPEN

**Remarks:** -

**See**

**Also:** OPEN



## CLOSEW# Graphic command

**Action:** closes a window.

**Syntax:** CLOSEW #n  
n: *ixp*

**Abbreviation:** -

**Explanation:** When in a graphic mode, CLOSEW #n closes a window previously opened with OPENW #n.

**Example:**

```
SCREEN 16 // EGA mode
OPENW #1,10,10,200,100,-1
REPEAT
  a$=inkey$
UNTIL a$=chr$(27)
CLOSEW #1
SCREEN 3
EDIT

// Draws a window on the screen. When the Esc key is
// pressed, the window is closed and the GFA-BASIC
// editor is invoked.
```

**Remarks:** -

**See**

**Also:** OPENW, TITLEW, INFOW, SIZEW, TOPW,  
FULLW, CLEARW

### CLR Command

**Action:** deletes all variables listed after this command.

**Syntax:** CLR x1[,x2,...]  
*x1,x2,...: variables of any type*

**Abbreviation:** -

**Explanation:** The variables in the list to be deleted with CLR must be separated by commas. The arrays cannot be deleted using CLR.

**Example:**

```
x=2
y=2
CLR x
PRINT x,y           // prints 0 2
```

**Remarks:** -

**See**

**Also:** CLEAR, ERASE

## CLS Graphic command

**Action:** clears the screen.

**Syntax:** CLS

**Abbreviation:** -

**Explanation:** Clears the screen by printing an ESC-E-CR. CLS can also be sent to files, in which case n is the channel number of the opened file.

**Example:**

```
DEFFILL 5
PBOX 10,10,100,100
REPEAT
UNTIL MOUSEK
CLS
```

```
// Draws a solid rectangle on the screen and deletes
// it when a mouse button is pressed.
```

**Remarks:** -

**See**

**Also:** -

### COLOR Command

**Action:** sets the character COLOUR.

**Syntax:** COLOUR *n*  
*n*: *iexp*

**Abbreviation:** col *n*

**Explanation:** COLOUR command sets the character COLOUR. The expression *n* specifies the COLOUR. Depending on the screen resolution it has the following meaning:

#### CGA resolutions:

##### 1. Palette

*n* = 1 turquoise

*n* = 2 violet

*n* = 3 white

##### 2. Palette

*n* = 1 green

*n* = 2 red

*n* = 3 yellow

#### EGA or VGA resolution:

*n* = 0 ( 0) black

*n* = 1 ( 1) blue

*n* = 2 ( 2) green

*n* = 3 ( 3) turquoise (cyan)

*n* = 4 ( 4) red

*n* = 5 ( 5) purple (magenta)

*n* = 6 (20) brown

*n* = 7 ( 7) lights grey

*n* = 8 (56) dark grey

*n* = 9 (57) light blue

*n* = 10 (58) light green

*n* = 11 (59) light turquoise

*n* = 12 (60) light red

*n* = 13 (61) light purple

*n* = 14 (62) light yellow

*n* = 15 (63) white

**Example:** COLOUR 15,3

**Remarks:** -

**See**

**Also:** SYSCOL

### COMBIN() Numeric function

**Action:** returns the number of combinations of  $n$  elements to  $k$ -th order without repetition.

**Syntax:** `COMBIN(n,k)`  
 $n, k$ : *iexp*

**Explanation:** `COMBIN(n,k)` is defined as:  
$$\text{COMBIN}(n,k) = n! / ((n-k)! * k!)$$

**Example:** `PRINT COMBIN(6,2)` // prints 15

**Remarks:** When  $k > n$  an error is reported.

**See**

**Also:** `FACT()`, `VARIAT()`

## CONT Command

**Action:**

1. a part of the SELECT...CASE structure. It performs a branch to the next CASE or DEFAULT.
2. an instruction which causes the program, previously interrupted with the STOP command, to continue.

**Explanation:**

re 1. see "SELECT...CASE Structure"

re 2. During program debugging it is often useful to interrupt the program at a particular place and inspect certain variables. The STOP command in a program halts the program at this point without deleting the variables. After such a STOP the program can be resumed with a CONT.

**Example:**

re 1. see "SELECT...CASE Structure"

re 2.

```
a%=15
b$="Hello"
c=2*9+7
PRINT a%,b$,c
STOP                                     // at this point the program
//                                     can resume execution with
//                                     a CONT.
a%*=c
PRINT a%
```

**Remarks:** -

**See**

**Also:** -

## COS() Trigonometrical function

**Action:** returns the cosine of a numeric expression.

**Syntax:** COS(*x*)  
*x*: *anexp*; angle in radians

**Explanation:** The cosine of an angle in a right-angled triangle corresponds to a quotient between the hypotenuse and the side forming the angle. When calculating COS(*x*) it is assumed that the value of *x* is given in radians.

**Example:**

```
PRINT COS(0)           // prints 1
PRINT COS(PI/2)        // prints 0
PRINT COS(PI)          // prints -1
PRINT COS(3*PI/2)      // prints 0
```

**Remarks:** COS() is the reverse function of ACOS().

**See**

**Also:** SIN(), SINQ(), COSQ(), TAN(), ASIN(), ACOS(), ATN(), ATAN()



## COSH() Transcendental function

**Action:** returns the hyperbolic cosine of a numeric expression.

**Syntax:** COSH(x)  
*x:* *anexp*; *x* = > 0

**Explanation:** The hyperbolic cosine applies to all real numbers greater than or equal to 0. It is obtained with the function:

$$\text{COSH}(x) = (\text{EXP}(x) + \text{EXP}(-x)) / 2$$

The function  $y = \text{COSH}(x)$  returns in  $y$  a real number greater than or equal to 1.

**Example:**  
PRINT COSH(2.14) // prints 4.30854...  
PRINT COSH(ARCOSH(2.14)) // prints 2.14

**Remarks:** COSH() is the reverse function of ARCOSH().

**See**

**Also:** SINH(), TANH(), ARSINH(), ARCOSH(),  
ARTANH()

## COSQ() Trigonometrical function

**Action:** returns the extrapolated cosine of a numeric expression.

**Syntax:** COSQ(x)  
*x: aexp; angle in degrees*

**Explanation:** For COSQ() GFA-BASIC uses an internal table with cosine values in one degree steps. COSQ(x) expects, therefore, the expression x to be in degrees. The intermediate values of function x are extrapolated in 1/16- degree steps. This accuracy is sufficient for plotting of graphs on the screen, particularly when there is no co-processor, since this function is several times faster than COS(x).

**Example:**

PRINT COSQ(180)	// prints	-1
PRINT COS(PI)	// prints	-1

**Remarks:** -

**See**

**Also:** SIN(), SINQ(), COS(), TAN(), ASIN(), ACOS(), ATN(), ATAN()

## CRSCOL Function

**Action:** returns the current cursor position.

**Syntax:** CRSCOL

**Abbreviation:** crsc

**Explanation:** returns the current cursor column.

**Example:** PRINT CRSCOL // prints the column of the  
// current cursor position

**Remarks:** -

**See**

**Also:** CRSLIN, POS()

### CRSLIN Function

**Action:** returns the current cursor position.

**Syntax:** CRSLIN

**Abbreviation:** crsl

**Explanation:** returns the current cursor line.

**Example:**

```
PRINT CRSLIN // prints the line of the
//          current cursor position
```

**Remarks:** -

**See**  
**Also:** CRSCOL, POS()

## CURVE Command

**Action:** draws a Bezier curve.

**Syntax:** CURVE x0,y0,x1,y1,x2,y2,x3,y3

**Abbreviation:** cur x0,y0,x1,y1,x2,y2,x3,y3

**Explanation:** CURVE x0,y0,x1,y1,x2,y2,x3,y3 draws a Bezier curve. The Bezier curve starts at x0,y0 and ends at x3,y3. At x0,y0 the curve is a tangent to the line from x0,y0 to x1,y1 and at x3,y3 a tangent to the line from x3,y3 to x2,y2.

If points x0,y0,...,x3,y3 are viewed as corners of a rectangle, the curve lies fully within this rectangle. (The curve can also be seen as a line between x0,y0 and x3,y3 which is pushed away from the points x1,y1 and x2,y2).

**Example:**

```
SCREEN 16           // EGA mode
CURVE 10,10,10,100,100,100,100,100
REPEAT
  a$=INKEY$
UNTIL a$=CHR$(27)
SCREEN 3
EDIT
```

// Draws a Bezier curve and waits for the Esc key.

**Remarks:** -

**See**

**Also:** -

### CVD() Function

**Action:** converts the first eight characters in a string from binary to IEEE double format.

**Syntax:** CVD(a\$)  
a\$: *sexp*

**Explanation:** The order of the bytes depends on the processor. For 80x86/8 or 8088 processors LSB (least significant byte) is converted first and MSB (most significant byte) is converted last.

**Example:**

```
OPEN "R",#1,"Test",19
FIELD #1,1 AS a$,2 AS b$,4 AS c$,4 AS d$,8 AS e$
a$=CHR$(123)
b$=MKI$((1234)
c$=MKL$(12345678)
d$=MKS$(1.23)
e$=MKD$(1.23)
//
PUT #1,1
GET #1,1
PRINT ASC(a$)'CVI(b$)'CVL(c$)'CVS(d$)'CVD(e$)

// prints
// 123 1234 12345678 123000019.. 123
```

**Remarks:** CVD() is the reverse function of MKD().

The statement CVS(MKS\$(...)) requires only half as much memory as CVD(MKD\$(...)), but it saves only about 7 places.

**See**

**Also:** ASC(), CVI(), CVL(), CVS(), CHR\$(), MKI\$(), MKL\$(), MKS\$(), MKD\$()

## CVI() Function

**Action:** converts the first two characters in a string to a 16 bit integer.

**Syntax:** CVI(a\$)  
*a\$: sexp*

**Explanation:** CVI takes the first two character in a string as a number. CVI(a\$) is equivalent to DPEEK(V:a\$). CVI returns 0 if the string length is less than two.

**Example:**

```
PRINT CVI("Hello GFA") // prints    24904
PRINT CVI(MKI$(24))    // prints    24

OPEN "R",#1,"Test",19
FIELD #1,1 AS a$,2 AS b$,4 AS c$,4 AS d$,8 AS e$
a$=CHR$(123)
b$=MKI$((1234)
c$=MKL$(12345678)
d$=MK$$(1.23)
e$=MKD$(1.23)
PUT #1,1
//
GET #1,1
PRINT ASC(a$)'CVI(b$)'CVL(c$)'CVS(d$)'CVD(e$)

// prints
// 123 1234 12345678 123000019.. 123
```

**Remarks:** CVI() is the reverse function of MKI\$().  
see CVD() function.

**See**

**Also:** ASC(), CVL(), CVS(), CVD(), CHR\$(), MKI\$(),  
MKL\$(), MKS\$(), MKD\$()

### CVL() Function

**Action:** converts the first four characters in a string to a 32 bit integer.

**Syntax:** CVL(a\$)  
*a\$: sexp*

**Explanation:** CVL takes the first four character in a string as a number. CVL(a\$) is equivalent to LPEEK(V:a\$). CVL returns 0 if the string length is less than four.

**Example:**

```
PRINT CVL("Hello GFA") // prints 1819042120
PRINT CVL(MKL$(123456)) // prints 123456

OPEN "R",#1,"Test",19
FIELD #1,1 AS a$,2 AS b$,4 AS c$,4 AS d$,8 AS e$
a$=CHR$(123)
b$=MKI$((1234)
c$=MKL$(12345678)
d$=MKS$(1.23)
e$=MKD$(1.23)
PUT #1,1
//
GET #1,1
PRINT ASC(a$)'CVI(b$)'CVL(c$)'CVS(d$)'CVD(e$)

// prints
// 123 1234 12345678 123000019.. 123
```

**Remarks:** CVL() is the reverse function of MKL\$().  
  
see CVD() function.

**See**

**Also:** ASC(), CVI(), CVS(), CVD(), CHR\$(), MKI\$(),  
MKL\$(), MKS\$(), MKD\$()



## CVS() Function

**Action:** converts the first four characters in a string from binary to IEEE single format.

**Syntax:** CVS(a\$)  
*a\$: sexp*

**Explanation:** CVS takes the first four character in a string as a number. CVS(a\$) is equivalent to SINGLE(V:a\$). CVS returns 0 if the string length is less than four.

**Example:**

```
PRINT CVS(MKS$(12.25)) // prints      12.25

OPEN "R",#1,"Test",19
FIELD #1,1 AS a$,2 AS b$,4 AS c$,4 AS d$,8 AS e$
a$=CHR$(123)
b$=MKI$((1234)
c$=MKL$(12345678)
d$=MKS$(1.23)
e$=MKD$(1.23)
PUT #1,1
//
GET #1,1
PRINT ASC(a$)'CVI(b$)'CVL(c$)'CVS(d$)'CVD(e$)

// prints
// 123 1234 12345678 123000019.. 123
```

**Remarks:** CVS() is the reverse function of MKS\$().  
  
see CVD() function.

**See Also:** ASC(), CVI(), CVL(), CVD(), CHR\$(), MKI\$(),  
MKL\$, MKS\$, MKD\$()

### DATA Command

**Action:** declares numerical and/or string constants.

**Syntax:** DATA k1[,k2,k3,...]  
*k1,k2,k3... numerical and/or string constants*

**Abbreviation:** dat k1[...]

**Explanation:** Using the DATA command, constant values and strings can be entered in an economical manner. The numerical values can be specified in decimal, hexadecimal, octal or binary format (see also VAL). Strings which contain commas must be enclosed in quotation marks. The DATA lines can contain mixed numerical and string constants.

The reading of this data into variables is performed with READ.

Both DATA and READ have a, so-called, data pointer. It always points to the next DATA value to be read with READ. When the program starts up the pointer always points the first value after the first DATA. The pointer can be set to specific DATA lines by using markers and the RESTORE command. A marker is a sequence of characters which can be composed of digits, letters, underline characters and dots. Each marker must end with a colon.

**Example:**

```
DATA 1,2,3,4,5,6,7,8,9,10
FOR i%=1 TO 10
    READ a%           // reads a value from the
//                   DATA line into the
//                   variable a%
    PRINT a%          // and prints it on the
//                   screen
NEXT i%
EDIT
```

```
DATA 10,&A,$A,&HA,&012,&X1010,%1010
FOR i%=1 TO 7
    READ a%
    PRINT a%
NEXT i%
EDIT                // reads the number 10
//                  seven times into the
//                  variable a% and prints
//                  them on the screen
```

```
DATA Günther, Peter, Klaus, Michael
FOR i%=1 TO 4
    READ a$
    PRINT a$''
NEXT i%
EDIT                // prints   Günther Peter
//                  Klaus Michael
```

```
DATA 1,"Hello World",2,"Goodbye World",3,"nice to see
you"
FOR i%=1 to 3
    READ a%,a$
    PRINT a%''a$''
NEXT i%
```

## Commands and functions

---

```
EDIT // prints 1 Hello World
// 2 Goodbye
// World 3 nice
// to see you
1a: // marker
DATA 1,2,3
2a: // marker
DATA Peter,Klaus,Michael
RESTORE 1a // position the data pointer
FOR i%=1 TO 3
  READ a%
  PRINT a%' '
NEXT i%
PRINT
//
RESTORE 2a // position the data pointer
FOR i%=1 TO 3
  READ a%
  PRINT a%
NEXT i%
EDIT // prints 1 2 3
// Peter Klaus Michael
```

**Remarks:**

-

**See**

**Also:**

READ, RESTORE

## DATE\$ Function

**Action:** returns the system date.

**Syntax:** DATE\$

**Explanation:** DATE\$ returns the system date in the following format:

DD.MM.YYYY (Day.Month.Year) or

MM.DD.YYYY (Month.Day.Year; US format)

The format is set with the MODE command.

**Example:** PRINT DATE\$                    // prints the system date

**Remarks:** -

**See**

**Also:** TIME\$

### DATE\$ = Command

**Action:** sets the system date.

**Syntax:** DATE\$ = date\$  
*date\$:* *sexp*

**Explanation:** DATE\$ = date\$ sets the system date in the following format:

DD.MM.YYYY (Day.Month.Year) or  
MM.DD.YYYY (Month.Day.Year; US format)

The format is set with the MODE command.

**Example:** DATE\$="01.10.1990"  
// or  
DATE\$="10.01.1990"

**Remarks:** The setting of both the system date and the system time can be done with the SETTIME command.

**See**

**Also:** TIME\$ =, SETTIME

## DEC Command

**Action:** decrements a numeric variable.

**Syntax:** DEC x  
x: *avar*

**Abbreviation:** -

**Explanation:** DEC x decrements the value of x by 1.

**Example:** x=2.7  
DEC x  
PRINT x // prints 1.7

**Remarks:** Although DEC can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of DEC

```
x=x-1
x:=x-1
x-=1
x--
SUB x,1 or
ADD x,-1
```

can be used also.

When integer variables are used DEC doesn't test for overflow!

**See**

**Also:** INC, ADD, SUB, MUL, DIV, ++, --, +=, -=, \*=, /=

### DEC\$( ) Function

**Action:** converts an integer expression to decimal representation.

**Syntax:** DEC\$(m[,n])  
*m,n: iexp*

**Abbreviation:** -

**Explanation:** DEC\$(m[,n]) converts the integer expression m into decimal representation. This is a base 10 number system with digits from 0 to 9. The optional parameter n specifies how many places should be used. If n is greater than the number of places needed by m, the number is padded with leading zeros.

**Example:**

```
PRINT DEC$(25)           // prints    25
PRINT DEC$(123,6)        // prints    000123
```

**Remarks:** -

**See**

**Also:** BIN\$( ), HEX\$( ), OCT\$( )



## DEFAULT Condition test

**Action:** a part of the SELECT...CASE structure. It contains a branch directive which is performed when either none of the CASE conditions is met or a CONT is used to explicitly branch to DEFAULT.

**Syntax:** DEFAULT

**Explanation:** see "SELECT...CASE" Structure

**Example:** -

**Remarks:** OTHERWISE and CASE ELSE are synonymous with DEFAULT and can be used instead.

**See**

**Also:** -

### DEFBIT Command

**Action:** declares specified variables as Boolean (bit) variables.

**Abbreviation:** defbi

**Syntax:** DEFBIT a\$  
*a\$: string constant*

**Explanation:** DEFBIT a\$ simplifies variable declaration. a\$ specifies the variables which should be declared as bit variables, i.e. as Boolean variables. a\$ can, for example, have the following forms:

a\$	Effect
"b"	All variables which begin with a 'b' are declared as Boolean variables.
"bo"	All variables which begin with a 'b' or 'o' are declared as Boolean variables.
"x-z"	All variables which begin with an 'x', 'y' or 'z' are declared as Boolean variables.
"b-d,x,-z"	All variables which begin with 'b' to 'd' and 'x' to 'z' are declared as Boolean variables.

Variable types can also be declared by appending the relevant postfix characters !, |, &, %, # or \$.

**Example:** DEFBIT "bo" // program

**Remarks:** -

**See**

**Also:** DEFBYT, DEFINT, DEFWRD, DEFFLT, DEFSTR, DEFSNG, DEFDBL

## DEFBYT Command

**Action:** declares specified variables as 1 byte (8 bit) integer variables.

**Abbreviation:** defby

**Syntax:** DEFBYT a\$  
a\$: *string constant*

**Explanation:** DEFBYT a\$ simplifies variable declaration. a\$ specifies the variables which should be declared as 1 byte integer variables, i.e. as 8 bit variables. a\$ ca, for example, have the following forms:

a\$	Effect
"b"	All variables which begin with a 'b' are declared as 1 byte integer variables.
"bo"	All variables which begin with a 'b' or 'o' are declared as 1 byte integer variables.
"x-z"	All variables which begin with an 'x', 'y' or 'z' are declared as 1 byte integer variables.
"b-d,x-z"	All variables which begin with 'b' to 'd' and 'x' to 'z' are declared as 1 byte integer variables.

Variable types can also be declared by appending the relevant postfix characters !, |, &, %, # or \$.

**Example:** DEFBYT "bo" // program

**Remarks:** -

**See**

**Also:** DEFBIT, DEFINT, DEFWRD, DEFFLT, DEFSTR, DEFSNG, DEFDBL

### DEFDBL Command

**Action:** declares specified variables as 8 byte (64 bit) floating point variables.

**Abbreviation:** defdb

**Syntax:** DEFDBL a\$  
*a\$: string constant*

**Explanation:** DEFDBL a\$ simplifies variable declaration. a\$ specifies the variables which should be declared as 8 byte floating point variables. a\$ can, for example, have the following forms:

a\$	Effect
"b"	All variables which begin with a 'b' are declared as 8 byte floating point variables.
"bo"	All variables which begin with a 'b' or 'o' are declared as 8 byte floating point variables.
"x-z"	All variables which begin with an 'x', 'y' or 'z' are declared as 8 byte floating point variables.
"b-d,x-z"	All variables which begin with 'b' to 'd' and 'x' to 'z' are declared as 8 byte floating point variables.

Variable types can also be declared by appending the relevant postfix characters !, |, &, %, # or \$.

**Example:** DEFDBL "bo" // program

**Remarks:**

**See**

**Also:**

DEFBIT, DEFBYT, DEFWRD, DEFINT, DEFFLT,  
DEFSTR, DEFSNG

### DEFFILL Graphic command

**Action:** defines a fill pattern.

**Syntax:** DEFFILL pattern  
*pattern:* *iexp*

**Abbreviation:** deff [pattern]

**Explanation:** DEFFILL defines a fill pattern for PBOX, PCIRCLE, PELLIPSE, POLYFILL and FILL graphic commands.

One the available dot or line patterns can be selected using the pattern option. (see Fill pattern table.

**Example:**

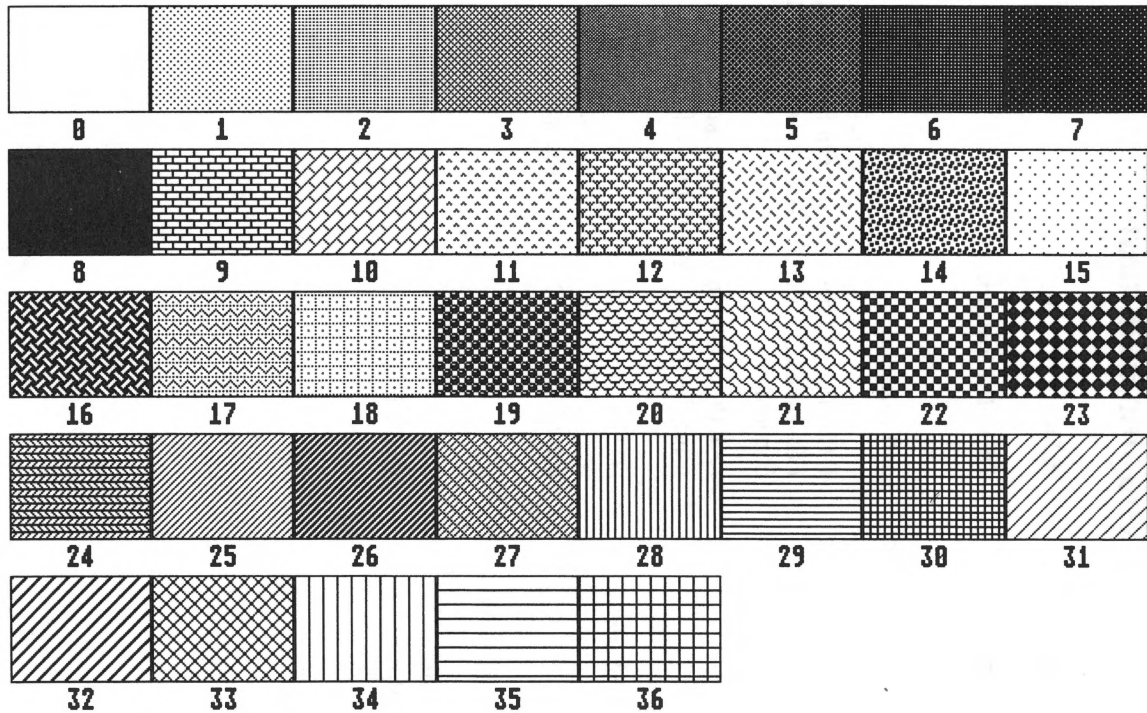
```
DEFFILL 8
PBOX 10,10,40,40 // box with a fill pattern
BOX 50,50,100,100 // box without a fill pattern
```

**Remarks:** -

**See**

**Also:** DEFLINE

# Table of Filling Pattern



## DEFFLT Command

**Action:** declares specified variables as 8 byte (64 bit) floating point variables.

**Abbreviation:** deffl

**Syntax:** DEFFLT a\$  
a\$: *string constant*

**Explanation:** DEFFLT a\$ simplifies variable declaration. a\$ specifies the variables which should be declared as 8 byte floating point variables. a\$ can, for example, have the following forms:

a\$	Effect
"b"	All variables which begin with a 'b' are declared as 8 byte floating point variables.
"bo"	All variables which begin with a 'b' or 'o' are declared as 8 byte floating point variables.
"x-z"	All variables which begin with an 'x', 'y' or 'z' are declared as 8 byte floating point variables.
"b-d,x-z"	All variables which begin with 'b' to 'd' and 'x' to 'z' are declared as 8 byte floating point variables.

Variable types can also be declared by appending the relevant postfix characters !, |, &, %, # or \$.

**Example:** DEFFLT "bo" // program



## Commands and functions 1490

---

**Remarks:** -

**See**

**Also:**

DEFBIT, DEFBYT, DEFWRD, DEFINT, DEFSTR,  
DEFSNG, DEFDBL

## DEFFN User defined function

**Action:** defines one-line functions.

**Syntax:** DEFFN name [parameterlist] = aexp  
*name:* *function name*  
*parameterlist:* *see Parameter list*  
*aexp:* *arithmetic expression*

or

DEFFN name\$ [parameterlist] = sexp  
*name\$:* *function name*  
*parameterlist:* *see Parameter list*  
*sexp:* *string expression*

**Abbreviation:** -

**Explanation:** The definition of the function, declared by DEFFN, is contained either in aexp or in sexp, depending on whether an arithmetic or a string expression is to be created.

DEFFN assumes that all variables used in parameterlist are local variables.

The function declared by DEFFN is invoked - like FUNCTION - by @functionname or FN functionname.

**Example:**

```
DEFFN nth_root(x,n%)=x^1/n%
PRINT "3rd root of PI = ";@nth_root1(PI,3)
DEFFN cut_string$(a$,p%)=LEFT$(a$,p%)
PRINT @cut_string$("Hello GFA",5) // prints 'Hello'
```

## Commands and functions

---

**Remarks:** -

**See**

**Also:**

PROCEDURE...RETURN,  
FUNCTION...Name...ENDFUNC,  
FUNCTION...Name\$...ENDFUNC

## DEFINT Command

**Action:** declares specified variables as 4 byte (32 bit) integer variables.

**Abbreviation:** defin

**Syntax:** DEFINT a\$  
*a\$: string constant*

**Explanation:** DEFINT a\$ simplifies variable declaration. a\$ specifies the variables which should be declared as 4 byte integer variables, i.e. 32 bit variables. a\$ can, for example, have the following forms:

a\$	Effect
"b"	All variables which begin with a 'b' are declared as 4 byte integer variables.
"bo"	All variables which begin with a 'b' or 'o' are declared as 4 byte integer variables.
"x-z"	All variables which begin with an 'x', 'y' or 'z' are declared as 4 byte integer variables.
"b-d,x-z"	All variables which begin with 'b' to 'd' and 'x' to 'z' are declared as 4 byte integer variables.

Variable types can also be declared by appending the relevant postfix characters !, |, &, %, # or \$.

**Example:** DEFINT "bo" // program

**Remarks:** -

**See**

**Also:** DEFBIT, DEFBYT, DEFWRD, DEFFLT, DEFSTR, DEFSNG, DEFDBL

### DEFLINE Graphic command

**Action:** defines the line type.

**Syntax:** DEFLINE [style], [thickness], [sstart,send]  
*style, thickness, sstart, send: iexp*

**Abbreviation:** defl [style], [thickness], [sstart,send]

**Explanation:** DEFLINE defines the appearance of a line drawn using the LINE, BOX, RBOX, CIRCLE, ELLIPSE and POLYLINE commands.

style defines the style of the line. The following options are possible:

- 0 -> solid line with background colour
- 1 -> solid line
- 2 -> dashed line with small spacing
- 3 -> dotted line
- 4 -> dotted and dashed line
- 5 -> dashed line with big spacing
- 6 -> -.-.-.-.

thickness refers to the thickness of the line in pixels. Only odd values are allowed as line thickness.

sstart and send define the start and end styles of a line. The following options are available:

- 0 -> square
- 1 -> arrow
- 2 -> round

The leading parameters can be omitted if commas separating the parameters are left in. DEFLINE „11 defines the start and end of a line as arrows but leaves the style and colour unchanged.

**Example:**

```
CLS
FOR i%=1 TO 6
  DECLINE i%
  LINE 50,i%*50,200,i%*50
NEXT i%
DECLINE 1,1,1,2
FOR i%=2 TO 12 STEP 2
  DECLINE ,i%
  LINE 250,i%*24,400,i%*25
NEXT i%
```

**Remarks:**

-

**See****Also:**

DEFFILL

### DEFNUM Command

**Action:** sets the formatting of output of numbers with all variations of PRINT.

**Syntax:** DEFNUM *n*  
*n*: *iexp*

**Abbreviation:** defn *n*

**Explanation:** All numbers following a DEFNUM command are output using *n* places (not counting the decimal point). The internal accuracy is not affected by this. The rounding off is performed using the *n* + 1-th position.

**Example:**

PRINT 100/3	// prints	33.3333333333
DEFNUM 5		
PRINT 100/3	// prints	33.333

**Remarks:** -

**See**

**Also:** STR\$(), PRINT USING, PRINT AT ... USING

## DEFSNG Command

**Action:** declares specified variables as 8 byte (64 bit) floating point variables.

**Abbreviation:** defsn

**Syntax:** DEFSNG a\$  
*a\$: string constant*

**Explanation:** DEFSNG a\$ simplifies variable declaration. a\$ specifies the variables which should be declared as 8 byte floating point variables. a\$ can, for example, have the following forms:

a\$	Effect
"b"	All variables which begin with a 'b' are declared as 8 byte floating point variables.
"bo"	All variables which begin with a 'b' or 'o' are declared as 8 byte floating point variables.
"x-z"	All variables which begin with an 'x', 'y' or 'z' are declared as 8 byte floating point variables.
"b-d,x-z"	All variables which begin with 'b' to 'd' and 'x' to 'z' are declared as 8 byte floating point variables.

Variable types can also be declared by appending the relevant postfix characters !, |, &, %, # or \$.

**Example:** DEFSNG "bo" // program



## Commands and functions

---

**Remarks:** -

**See**

**Also:** DEFBIT, DEFBYT, DEFWRD, DEFINT, DEFFLT,  
DEFSTR, DEFDBL

## DEFSTR Command

**Action:** declares specified variables as string variables.

**Abbreviation:** defst

**Syntax:** DEFSTR a\$  
*a\$: string constant*

**Explanation:** DEFSTR a\$ simplifies variable declaration. a\$ specifies the variables which should be declared as 4 byte integer variables, i.e. 32 bit variables. a\$ can, for example, have the following forms:

a\$	Effect
"b"	All variables which begin with a 'b' are declared as string variables.
"bo"	All variables which begin with a 'b' or 'o' are declared as string variables.
"x-z"	All variables which begin with an 'x', 'y' or 'z' are declared as string variables.
"b-d,x-z"	All variables which begin with 'b' to 'd' and 'x' to 'z' are declared as string variables.

Variable types can also be declared by appending the relevant postfix characters !, |, &, %, # or \$.

**Example:** DEFSTR "bo" // program

**Remarks:** -

**See**

**Also:** DEFBIT, DEFBYT, DEFWRD, DEFINT, DEFFLT, DEFSNG, DEFDBL

### DEFWRD Command

**Action:** declares specified variables as 2 byte (16 bit) integer variables.

**Abbreviation:** defwr

**Syntax:** DEFWRD a\$  
*a\$: string constant*

**Explanation:** DEFWRD a\$ simplifies variable declaration. a\$ specifies the variables which should be declared as 2 byte integer variables, i.e. 16 bit variables. a\$ can, for example, have the following forms:

a\$	Effect
"b"	All variables which begin with a 'b' are declared as 2 byte integer variables.
"bo"	All variables which begin with a 'b' or 'o' are declared as 2 byte integer variables.
"x-z"	All variables which begin with an 'x', 'y' or 'z' are declared as 2 byte integer variables.
"b-d,x-z"	All variables which begin with 'b' to 'd' and 'x' to 'z' are declared as 2 byte integer variables.

Variable types can also be declared by appending the relevant postfix characters !, |, &, %, # or \$.

**Example:** DEFWRD "bo" // program

**Remarks:** -

**See**

**Also:** DEFBIT, DEFBYT, DEFINT, DEFFLT, DEFSTR, DEFSNG, DEFDBL

## DEG() Trigonometrical function

**Action:** returns the angle of a numeric expression in degrees.

**Syntax:** DEG(x)  
*x: aexp; angle in radians*

**Explanation:** DEG(x) is equivalent to  $x * 180 / \text{PI}$

**Example:**

PRINT DEG(PI/2)	// prints	90
PRINT DEG(PI)	// prints	180
PRINT DEG(3*PI/2)	// prints	270
PRINT DEG(2*PI)	// prints	360

**Remarks:** DEG(x) is the reverse function of RAD(x), which means:

$$\text{DEG}(\text{RAD}(\text{PI})) = \text{PI} = 3.14\dots$$

**See**

**Also:** RAD()

### DELAY Command

**Action:** interrupts a program

**Syntax:** DELAY a  
*a: aexp*

**Abbreviation:** -

**Explanation:** DELAY a interrupts a program for 'a' seconds. In contrast to a FOR...NEXT waiting loop, this command is independent of the processor frequency.

**Example:** DELAY 120 // a two minute pause  
// (a lunch break at GFA!)

**Remarks:** In contrast to PAUSE (dependent on the operating system) the time specified with DELAY is portable. Naturally, DELAY uses PAUSE internally.

**See**

**Also:** PAUSE

## DELETE Command

**Action:** deletes an element from a one-dimensional array of any variable type.

**Syntax:** DELETE x(m)  
*m:* *iexp*  
*x():* *one-dimensional array of any variable type*

**Abbreviation:** del x(m)

**Explanation:** DELETE x(m) deletes the element indexed by m from the array x(). In other words all array items whose indices are greater than or equal to m are shifted one position up. The last element in the array is deleted (with 0 or "" depending on type).

**Example:**

```
DIM a$(4)
a$(1)="GFA BASIC MS DOS"
a$(2)="GFA BASIC OS/2"
a$(3)="MS PDS 7.0"
a$(4)="GFA BASIX"
DELETE a$(3)
FOR i%=1 to 4
    PRINT a$(i%)
NEXT i%
```

```
// prints
// GFA BASIC MS DOS,
// GFA BASIC OS/2
// GFA BASIX
```

**Remarks:** -

**See**

**Also:** INSERT

### DFREE() Function

**Action:** determines the amount of free disk space on current drive.

**Syntax:** DFREE(*n*)  
*n*: *iexp*

**Explanation:** DFREE (disk free) returns the amount of free disk space on current drive. The current drive can be set using the CHDRIVE command. If the current drive is a hard disk partition, DFREE() can take a substantial amount of time.

*n* can assume the values from 0 to 26, whereby *n*=0 always refers to the current drive. The *n* values from 1 to 26 correspond to drives A to Z.

**Example:**

```
CHDRIVE "E:\"           // sets the partition E as
//                      the current drive
PRINT DFREE(0)          // determines the amount of
//                      free disk space in
//                      partition E
PRINT DFREE(1)          // determines the amount of
//                      free disk space on drive A
```

**Remarks:** -

**See**

**Also:** -

## DIM Command

**Action:** creates an array of any variable type

**Syntax:** DIM x1(n1[,n2,...,n6])[,x2(n1[,n2,...,n6]),...]  
*x1,x2,...: arrays of any variable type*  
*n1,n2,...,n6: iexp*

**Abbreviation:** -

**Explanation:** DIM is used to define numeric and/or string arrays of up to 6 dimensions. Only variables of the same type are allowed in one array. The elements are addressed using indices.

The number of elements in one-dimensional arrays is only limited by available memory. In multidimensional arrays the number of elements for the first dimension is also "unlimited". However, the size of the first dimension as well as the product of all other dimensions must be less than 65535.

For OPTION BASE 0, the indexing of array elements in all dimensions begins with element 0. For example, DIM a&(19) will define an integer array with 20 2-byte variables: a&(0) .... a&(19). DIM a(5,5) will define a floating point array with 36 double variables: a(0,0), a(0,1),... a(5,5).

If several arrays are defined simultaneously they must all be listed after the DIM command and separated by commas.



## Commands and functions

---

### Example:

Assuming OPTION BASE 0 has been selected:

`DIM a%(19,9,2,13)` is allowed, since the number of dimensions is less than 6 and the number of elements in dimensions 2 to 4 is 420, which is less than 65535.

`DIM a%(99,99,99,9)` is not allowed, since the number of elements in dimensions 2 to 4 is 100000, which is greater than 65535.

`DIM a(10,10),a%(3),a$(14,2,3)` defines a two-dimensional floating point array, a one-dimensional integer array and one three-dimensional string array.

### Remarks:

-

### See

### Also:

`DIM?()`, `ERASE`

## DIM?() Function

**Action:** returns the number of elements in a numeric array or the number of strings in a string array.

**Syntax:** DIM?(x())  
*x(): array of any variable type*

**Example:** DIM a%(19,9,2,13)  
PRINT DIM?(a%()) // prints 8400

**Remarks:** -

**See**

**Also:** DIM, ERASE

### DIR\$() Function

**Action:** sets the current path name.

**Syntax:** DIR\$(*n*)  
*n*: *iexp*

**Explanation:** DIR\$(*n*) sets the current path name for the drive previously set with CHDIR.

*n* can assume the values from 0 to 26, whereby *n*=0 always refers to the current drive. The *n* values from 1 to 26 correspond to drives A to Z.

**Example:**

```
PRINT DIR$(0)      // prints the path name
//                for the current drive
//
PRINT DIR$(2)      // prints the path name for
//                the drive B
```

**Remarks:** -

**See**

**Also:** CHDIR

## DIR...TO Command

**Action:** prints the directories in current path name.

**Syntax:** DIR path\$ [TO file\$]  
*path\$:* *sexp; current path name*  
*file\$:* *sexp; optional file name*

**Abbreviation:** -

**Explanation:** DIR path\$ returns the directories in path name specified in path\$. If path\$ ends with a "." or "\", GFA-BASIC automatically appends ".\*". The default destination for the output of the directory is to screen. The specification of TO file\$ is optional. It can be used to redirect the directory output to a file or a peripheral device.

**Example:**

```
path$="A:\TEST\A1:"
DIR path$                // prints the directory of
//                      A:\TEST\A1 to the
//                      screen
//
DIR path$ TO "A:\VERZ.TXT" // writes the directory of
//                      A:\TEST\A1 to the file
//                      A:\VERZ.TXT
//
DIR path$ TO "PRT:"      // prints the directory of
//                      A:\TEST\A1 to the
//                      printer
//
DIR path$+"*.TXT" TO "PRN" // prints all files
//                      with extention TXT in
//                      A:\TEST\A1 to the
//                      printer
```

## Commands and functions

---

**Remarks:** -

**See**

**Also:** FILES ... TO

## DIV Command

**Action:** divides a numeric expression into a numeric variable.

**Syntax:** DIV x,y  
x: *avar*  
y: *aexp*

**Abbreviation:** -

**Explanation:** DIV x,y divides the expression y into the value in variable x.

**Example:** x=126  
DIV x,2+1  
PRINT x // prints 42

**Remarks:** Although DIV can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of DIV x,y

x = x/y or  
x := x/y or  
x /= y

can be used also.

When integer variables are used DIV doesn't test for overflow!

**See**

**Also:** DEC, INC, ADD, SUB, MUL, ++, --, +=, -=, \*=, /=

### DIV() Function

**Action:** divides two integer expressions.

**Syntax:** DIV(*i*,*j*)  
*i*,*j*: *iexp*

**Explanation:** DIV(*i*,*j*) calculates the quotient of integer expressions *i* and *j*, and, optionally, writes the result to a variable.

**Example:** PRINT DIV(126,SUCC(2)) // prints 42  
1%=DIV(126,SUCC(2))

PRINT 1% // prints 42

**Remarks:** The ADD(), SUB(), MUL() and DIV() functions can be mixed freely with each other. For example:

1%=SUB(5^3,20/4+3) or

1%=SUB(5^3,ADD(DIV(20,4),3))

**See**

**Also:** ADD(), SUB(), MUL(), MOD(), PRED(), SUCC()

## DO...LOOP Structure

**Action:** an infinite programming loop, which runs "endlessly" and can only be interrupted by the command EXIT IF.

**Syntax:** DO  
          // programsegment  
LOOP

**Abbreviation:** -

**Explanation:** DO...LOOP is an endless loop which can only be terminated by the command EXIT IF. In most cases it contains the complete program.

The most common usage of DO...LOOP is in connection with the EXIT IF command:

The WHILE...WEND loop always tests the condition before running, while REPEAT...UNTIL loop always tests the condition after running. By using EXIT IF the loop can be interrogated at any point to determine whether the condition is logically "true". If the condition after EXIT IF is logically "true" the program will always branch to the line immediately following the next loop end.

**Example:**

```
DO
  r=0
  INPUT "Enter radius";r
  EXIT IF r<0
  PRINT "The circumference of the circle is :
    ";2*PI*r
LOOP
PRINT "End of program!"
```



## Commands and functions

---

```
// The program requests the user to enter the radius
// of a circle. If the entered value is greater than
// or equal to zero, the circumference of the circle
// is calculated and displayed.
// You are then requested to enter another value. If
// you enter a negative value the loop is terminated
// and "End of program!" is displayed.
```

### Remarks:

The DO...LOOP loop is the most universal programming loop and it can be used to emulate all other loops:

#### Example:

```
i%=0                FOR i%=1 TO n%
DO                  // programsegment
    EXIT IF i%>n%    NEXT i%
    //programsegment
LOOP

DO                  WHILE INKEY$<>"A"
    EXIT IF INKEY$="A" // programsegment
    // programsegment WEND
LOOP

DO                  REPEAT
    // programsegment // programsegment
EXIT IF INKEY$="A"  UNTIL INKEY$="A"
LOOP
```

Even more powerful loop conditions can be created by combining the DO...LOOP loop with the evaluation part of the WHILE...WEND and/or REPEAT...UNTIL loops:

```
DO UNTIL EOF
  INPUT #1,a$
  b$=LEFT$(TRIM$(a$),1)
LOOP WHILE UPPER$(b$)="A"

// Reads a sequential character string from channel
// 1, until the end of file (UNTIL condition) or
// until the character string starts with something
// other than lowercase or uppercase "a".

DO WHILE (MOUSEX AND 1)
  BOX MOUSEX,MOUSEY,ADD(MOUSEX,10),ADD(MOUSEY,10)
LOOP UNTIL UPPER$(INKEY$)="A"

// When the left mouse button is pressed it draws a
// rectangle at the current mouse position, until a
// lowercase or uppercase "a" is typed on the key-
// board.
```

The following loop combinations are possible:

```
DO ... LOOP
DO ... LOOP UNTIL
DO ... LOOP WHILE
DO ... WEND
DO ... UNTIL
```

WHILE ... LOOP	DO WHILE ... LOOP
WHILE ... LOOP UNTIL	DO WHILE ... LOOP UNTIL
WHILE ... LOOP WHILE	DO WHILE ... LOOP WHILE
WHILE ... WEND	DO WHILE ... WEND
WHILE ... UNTIL	DO WHILE ... UNTIL

REPEAT ... LOOP	DO UNTIL ... LOOP
REPEAT ... LOOP UNTIL	DO UNTIL ... LOOP UNTIL
REPEAT ... LOOP WHILE	DO UNTIL ... LOOP WHILE
REPEAT ... WEND	DO UNTIL ... WEND
REPEAT ... UNTIL	DO UNTIL ... UNTIL

## Commands and functions 8488

---

```
DO ... LOOP
DO ... LOOP UNTIL
DO ... LOOP WHILE
DO ... WEND
DO ... UNTIL
```

**See**

**Also:** [FOR...NEXT](#), [WHILE...WEND](#), [DO...LOOP](#)

## DOUBLE{} Function

**Action:** reads a 64-bit IEEE double number from an address.

**Syntax:**       DOUBLE{}  
                  *addr: address*

**Explanation:**   Reads a 64-bit IEEE double number from an address.

**Example:**

```
DIM a(30)
FOR i%=0 TO 30
  a(i%)=random(100)
NEXT i%
@test(V:a(27))
//
PROCEDURE test(addr%)
  LOCAL x
  x=DOUBLE{addr%}
  x+                // or any other operation
  DOUBLE {addr%}=x
RETURN

// Simulates a VAR parameter for an array element, in
// this case for a(27).
```

**Remarks:**       addr: see the {} function.

**See**

**Also:**            BYTE{}, CARD{}, WORD{}, INT{}, LONG{}, {},  
                  SINGLE{}

### DOUBLE{} Command

**Action:** writes a 64-bit IEEE double number to an address.

**Syntax:** `DOUBLE{} = x`  
*addr: address*

**Explanation:** Writes a 64-bit IEEE double number to an address.

**Example:**

```
DIM a(30)
FOR i%=0 TO 30
    a(i%)=random(100)
NEXT i%
@test(V:a(27))
//
PROCEDURE test(addr%)
    LOCAL x
    x=DOUBLE{addr%}
    x+                // or any other operation
    DOUBLE {addr%}=x
RETURN

// Simulates a VAR parameter for an array element, in
// this case for a(27).
```

**Remarks:** addr: see the {} function.

**See**

**Also:** `BYTE{}`, `CARD{}`, `WORD{}`, `INT{}`, `LONG{}`, `{}`, `SINGLE{}`

## DPEEK() Function

**Action:** reads two bytes (16 bits) from an address.

**Syntax:** DPEEK(addr)  
*addr: address*

**Explanation:** Reads two bytes (16 bits) from an address.

**Example:** PRINT DPEEK(\*a) // prints the first word of  
// the contents of a

**Remarks:** WORD{} and INT{} are synonymous with DPEEK()  
and can be used instead.  
addr: see the {} function.

**See**

**Also:** PEEK(), BYTE{}, WORD{}, INT{}, LPEEK(),  
LONG{}

### DPOKE Command

**Action:** writes a word (16 bits) to an address.

**Syntax:** DPOKEaddr,m  
*addr: address*  
*m: iexp*

**Explanation:** Writes a word (16 bits) to an address.

**Example:** -

**Remarks:** WORD{ } = and INT{ } = are synonymous with DPOKE() and can be used instead.

addr: see the { } function.

**See**

**Also:** POKE(), BYTE{ } =, WORD{ } =, INT{ } =, LPOKE, LONG{ } =

## DRAGBOX Graphic function

**Action:** moves a graphic segment around the screen.

**Syntax:** DRAGBOX x1,y1,w1,h1[,x2,y2,w2,h2],x3,y3  
*x1,y1,w1,h1,x2,y2,w2,h2,x3,y3: iexp*

**Abbreviation:** dra x1,y1,w1,h1[,x2,y2,w2,h2],x3,y3)

**Explanation:** DRAGBOX() works only if the graphic mode is on. This function creates a rectangular cut-out with the width w1 and height h1, whose upper left corner is specified with x1 and y1.

This rectangle can be moved within another rectangle by holding down the left mouse button and moving the mouse. The upper left corner of the second rectangle is given in x2 and y2, the width in w2 and the height in h2. When the movement is finished x3 and y3 contain the coordinates of the upper left corner of moved rectangle.

**Example:**

```
SCREEN 16           // EGA mode
GRAPHMODE 3        // XOR
OPENW #1,10,10,400,200,-1
x1%=20
y1%=20
w1%=70
w2%=35
x2%=10
w2%=_X             // horizontal window extension in pixels
//
h2%=_Y             // vertical window extension in pixels
//
DO
  IF MOUSEK AND 1
    DRAGBOX 20,20,100,100,x2%,y2%,w2%,h2%,x3%,y3%
```



## Commands and functions

---

```
x1%=x3%
y1%=y3%
BOX x1%, y1%,ADD(x1%,w1%),ADD(y1%,h1%)
ENDIF
UNTIL MOUSEK AND 2
CLOSEW #1
SCREEN 3
EDIT

// Draws and moves a DRAGBOX.
```

**Remarks:**

-

**See**

**Also:** RUBBERBOX

## DRAW Command

**Action:** draws a point or a line between two points on the screen.

**Syntax:** DRAW [T0] [x,y] or  
DRAW [x1,y1] [T0 x2,y2][T0 x3,y3]... or  
DRAW exp or  
DRAW(i) or  
SETDRAW

*x,y,x1,y1,x2,y2,i: iexp  
exp: a mixture of sexp and aexp,  
whereby the first expression  
must be a sexp. The individual  
exps are separated by a comma,  
semi-colon or apostrophe.*

**Abbreviation:** dr [T0] [x,y] or  
dr [x1,y1] [T0 x2,y2][T0 x3,y3]... or  
dr exp or  
dr(i) or  
setd

**Explanation:** DRAW x,y is equivalent to the PLOT command, that is, a point with the coordinates x,y is drawn on the screen. DRAW TO x,y draws a line between the point with the coordinates x,y and the last set point. It is irrelevant whether this point was set with PLOT, LINE or DRAW.

DRAW x1,y1 TO x2,y2 is equivalent to the LINE command. However, additional coordinates can also be added. It is therefore possible to draw polygons in this manner.

## Commands and functions

---

DRAW exp enables definition of commands similar to certain LOGO graphic commands (turtle graphics) or HPGL Hewlett-Packard standard plotter language commands. It is possible, in this way, to move an imaginary pencil across the screen, drawing as needed. The parameters for individual commands are floating point numbers which can also be specified using strings. The following commands are available:

<b>FD n Forward</b>	moves the 'pencil' n pixels 'forward'.
<b>BK n Backward</b>	moves the 'pencil' n pixels 'backwards'.
<b>SX x Scale X</b>	scales the 'pencil movement' for FD
<b>SY y Scale Y</b>	or BK by the factor given in x or y. The scaling can be turned off with SX 0 or SY 0.
<b>LT w Left Turn</b>	turns the 'pencil' left by the angle w (in degrees).
<b>RT w</b>	the same to the right
<b>TT w Turn To</b>	moves the 'pencil' to an absolute angle (in degrees). The assignment is as follows: <b>w = 0:</b> up or north <b>w = 90:</b> right or east <b>w = 180:</b> down or south <b>w = 270:</b> left or west
<b>MA x,y Move Absolute</b>	moves the 'pencil' to absolute coordinates x and y.
<b>DA x,y Draw Absolute</b>	moves the 'pencil' to absolute coordinates x and y, and then draws a line in current colour from the last set position to point (x,y).
<b>MR x,y Move Relative</b>	like MA, except that it moves relative to last position.

<b>DR x,y Draw Relative</b>	like MR, except that it moves relative to last position.
<b>CO n Colour</b>	defines colour n as drawing colour.
<b>PU Pen Up</b>	lifts the 'pencil' up.
<b>PD Pen Down</b>	lowers the 'pencil' down.

**DRAW(i)** is a function which, depending on i, returns the following values:

**i = 0** x coordinate (floating point number)  
**i = 1** y coordinate (floating point number)  
**i = 2** angle in degrees (floating point number)  
**i = 3** scaling on the x axis (floating point number)  
**i = 4** scaling on the y axis (floating point number)  
**i = 5** pen status (-1 for PD and 0 for PU)

**SETDRAW** sets various values in the **DRAW** exp command. For example, **SETDRAW x,y,w** is equivalent to **DRAW "MA",x,y"TT",w** command.

### Example:

```
CLS
DRAW 100,100           // sets a point at 100,100
DRAW TO 10,10          // draws a line from 100,00
                        // to 10,10
                        //
DRAW 10,10 TO 20,20 TO 30,30 // draws a line
                        // line from 10,10 to 20,20
                        // and from 20,20 to 30,30
                        //
DRAW "ma 160,200 tt0" // starts at 160,200 with
                        // angle 0
                        //
FOR i%=3 TO 10
    @corner(i%,90)      // draws a polygon with i
NEXT i%                // corners
                        //
```

## Commands and functions

---

```
PROCEDURE corner(n%,r%)
  LOCAL i%
  FOR i%=1 TO n%
    DRAW "fd",r%,"rt",360/n%
  NEXT i%
RETURN
//
CLS
FOR i%=0 TO 359
  SETDRAW 320,200,i%
  DRAW "fd 45 rt 90 fd 45 rt 90 fd 45 rt 90 fd 45"
  DRAW "bk 90 rt 90 bk 90 rt 90 bk 90 rt 90 bk 90"
  DRAW "fd 45 rt 90 fd 45 rt 90 fd 45 rt 90 fd 45"
  DRAW "bk 90 rt 90 bk 90 rt 90 bk 90 rt 90 bk 90"
NEXT i%

// Draws a small and a large rectangle which both
// rotate around their own axis.
```

**Remarks:** -

**See**

**Also:** PLOT, LINE

## EAVAIL Command

**Action:** returns the number of free Expanded Memory (EMS) pages.

**Syntax:** EAVAIL *n*  
*n:* *ivar*

**Abbreviation:** -

**Explanation:** The EMS is segmented into 16 K pages. It is recommended to check the amount of available EMS memory with EAVAIL before invoking the GFA-BASIC commands EPUSH, EMEMPUSH, EPARLOAD.

**Example:** EAVAIL x%  
PRINT x%

// Returns the number of free EMS memory pages.

**Remarks:** -

**See**  
**Also:** EPUSH, EPOP, EGET, EKILL, EDIR,  
EPARLOAD, EPARSAVE, EMEMPUSH,  
EMEMPOP, EMEMGET

## EDIR Command

**Action:** redirection of Expanded Memory (EMS) commands to files

**Syntax:** EDIR ["pathname"]

**Abbreviation:** -

**Explanation:** If EMS is not available or a different medium should be used instead of EMS memory, EDIR can indicate a drive and a directory which are treated by EMS commands as Expanded Memory. The file name is a hexadecimal number (its stack, counting from below and starting with 1), while the extension is .\$@@. EDIR without a pathname reverts back to the EMS.

**Example:** EDIR "C:\EMSSIM\"

**Remarks:** -

**See**

**Also:** EPUSH, EPOP, EGET, EKILL, EAVAIL, EPARLOAD, EPARSAVE, EMEMPUSH, EMEMPOP, EMEMGET

### EDIT Command

**Action:** returns to the GFA-BASIC editor without deleting the program.

**Syntax:** EDIT

**Abbreviation:** ed

**Explanation:** EDIT returns from uncompiled programs back to the GFA-BASIC editor. All file channels are thereby closed.

**Example:**

```
CLS
SCREEN 16           // EGA mode
OPENW #1,10,10,400,200,-1
FOR i%=1 TO 10
    PRINT i%
NEXT i%
INTR($16,_AH=0)
EDITOR
```

```
// Turns the graphic mode on and opens a window. The
// digits from 1 to 10 are then written in the window
// and the program waits for a keypress. If the key
// is pressed the text mode is turned back on and the
// GFA-BASIC editor is invoked.
```

**Remarks:** -

**See**

**Also:** END, SYSTEM, QUIT



## EGET Command

**Action:** reads variables from Expanded Memory (EMS)

**Syntax:** EGET a:"NAME", a: number,a():"name",a():number,...  
*a:* variable of any type (*a%*,*a&*,...)  
*a():* array with variables of any type (*a()*,  
*a\$( ),...*)  
*name:* *sexp*; the name assigned with EPUSH  
*number* *iexp*

**Abbreviation:** -

**Explanation:** EGET a:"name",a:number,a():"name",a():number disregards the stack organization of EMS memory (see EPOP). Because of this, it is possible to access any variable or any array within the EMS memory. The access is by a number, which describes the position of the variable or array on stack, or by a name, which was given with EPUSH. The variables and arrays read from EMS memory with EGET are not deleted and can - in contrast to EPOP - be accessed several times.

The variables are erased from EMS by using the EKILL command (refer to the command itself).

The usage of names makes particular sense when one program stores and another program retrieves the data from EMS memory. If one program changes the EMS stack position the second should not modify it or the names will become irrelevant.

When using numbers with EGET, the negative numbers access the stack from the top (EGET a:0 gets the last saved variable or array, while EGET a:-1 the one saved before the last). The positive numbers access the stack from the bottom (EGET a:1 gets the first saved variable or array).

## Commands and functions

---

### Example:

```
SCREEN 3
n%=500
DIM a%(n%,1)
//
FOR i%=0 TO n%
  a%(i%,0)=i%
  a%(i%,1)=n%+i%
NEXT i%
EPUSH a%()
DIM a(n%)
x=SINQ(23)
FOR i%=1 TO n%
  a(i%)=x
NEXT i%
EPUSH a()
//
EGET b():0
EGET c%():-1
FOR i%=0 TO n%
  PRINT c%(i%,0)'
  PRINT c%(i%,1)'
NEXT i%
```

```
// The integer array a%() is dimensioned and
// initialized with digits from 0 to n% and from n%
// to 2*n%. This array is then stored in EMS memory.
// Another array a(), this time floating point, is
// then dimensioned, initialized with a constant and
// also stored in EMS memory. EGET b():0 then loads
// this last array into array b(), which does not
// have to be re-dimensioned.
// EGET c%():-1 reads the array a%() from EMS and
// into array c%().
```

**Remarks:**

-

**See**

**Also:**

EPUSH, EPOP, EDIR, EAVAIL, EKILL,  
EPARLOAD, EPARSAVE, EMEMPUSH,  
EMEMPOP, EMEMGET

### EKILL Command

**Action:** deletes variables and arrays from Expanded Memory (EMS).

**Syntax:** EKILL *n*  
*n*: *iexp*

**Abbreviation:** -

**Explanation:** The variables and arrays stored in EMS with EPUSH are not deleted when loaded back into main memory with EGET. EKILL *n* deletes *n* variables in EMS memory. An array counts as a single variable. Since EMS variables are arranged as a stack (First In Last Out), EKILL *n* deletes the last *n* variables stored in EMS. EKILL without parameters deletes the whole EMS stack.

**Example:**

```
SCREEN 3
n%=500
DIM a%(n%,1)
//
FOR i%=0 TO n%
  a%(i%,0)=i%
  a%(i%,1)=n%+i%
NEXT i%
EPUSH a%()
DIM a(n%)
x=SINQ(23)
FOR i%=1 TO n%
  a(i%)=x
NEXT i%
EPUSH a()
//
EGET b():0
EGET c%():-1
```

```
FOR i%=0 TO n%
  PRINT c%(i%,0)'
  PRINT c%(i%,1)'
NEXT i%
//
EKILL 2

// The integer array a%() is dimensioned and
// initialized with digits from 0 to n% and from n%
// to 2*n%. This array is then stored in EMS memory.
// Another array a(), this time floating point, is
// then dimensioned, initialized with a constant and
// also stored in EMS memory.
// EGET b():0 then loads this last array into array
// b(), which does not have to be re-dimensioned.
// EGET c%():-1 reads the array a%() from EMS and
// into array c%().
// EKILL 2 deletes both arrays from the EMS stack.
```

**Remarks:**

-

**See**

**Also:**

EPUSH, EPOP, EGET, EDIR, EAVAIL,  
EPARLOAD, EPARSAVE, EMEMPUSH,  
EMEMPOP, EMEMGET

### ELLIPSE Graphic command

**Action:** draws an ellipse.

**Syntax:** ELLIPSE *x,y,rx,ry,w1,w2*  
*x,y,rx,ry,w1,w2: iexp*

**Abbreviation:** el *x,y,rx,ry,w1,w2*

**Explanation:** ELLIPSE *x,y,rx,ry,w1,w2* draws an ellipse with the horizontal radius *rx* and the vertical radius *ry*, around the centre point with coordinates *x,y*.

The start (*w1*) and end (*w2*) angles can optionally be specified to define an ellipse arc. The angles *w1* and *w2* are given whole degree steps.

**Example:** CLS  
ELLIPSE 320,200,200,100,90,180

**Remarks:** -

**See**

**Also:** CIRCLE, PCIRCLE, PELLIPSE

## EMEMGET Command

**Action:** reads contents of Expanded Memory (EMS) back into MS-DOS RAM

**Syntax:** EMEMGET addr,size[:"NAME"] or  
EMEMGET addr,size[:number]  
*addr:*            *address*  
*size:*            *iexp*  
*number:*        *iexp*

**Abbreviation:** -

**Explanation:** EMEMGET addr,size["name"] and EMEMGET addr,size[:number] work like EGET, except that, instead of variables and arrays, whole areas of EMS memory are read into MS-DOS RAM. This enables, for example, reading of portions of an array stored in EMS back into main memory.

addr specifies the starting address, while size specifies the amount of MS-DOS memory, which will be loaded with the contents of EMS memory.

name describes an EMS area previously loaded with EMEMPUSH.

number specifies, like with EGET, the position of this area on stack.

**Example:**

```
DIM a(100,100)
FOR i%=1 TO 100
  FOR j%=1 TO 100
    a(i%,j%)=j%
  NEXT j%
NEXT i%
EMEMPUSH V:a(0,0),101 << 3,"1.Column"
ERASE a()
```

## Commands and functions

---

```
DIM b(50)
EMEMGET V:b(0),51 << 3,"1.Column"
FOR i%=1 TO 100
  PRINT b(i%)'
NEXT i%
```

```
// Dimensions the floating point array a() and fills
// it with numbers from 0 to 100, column-wise. The
// first column in a() is then moved to EMS and a()
// is deleted. Array b() is then dimensioned and the
// first 51 values in first column of former array
// a() are moved to b().
```

**Remarks:** addr: see the {} function.

**See**

**Also:**

EPUSH, EPOP, EGET, EKILL, EDIR, EAVAIL,  
EPARLOAD, EPARSAVE, EMEMPUSH,  
EMEMPOP



## EMEMPOP Command

**Action:** reads contents of Expanded Memory (EMS) back into MS-DOS RAM.

**Syntax:** EMEMPO *addr,size*  
*addr: address*  
*size: iexp*

**Abbreviation:** -

**Explanation:** EMEMPOP *addr,size* works like EPOP, except that, instead of variables and arrays, whole areas of EMS memory are read into MS-DOS RAM. This enables, for example, reading of portions of an array stored in EMS back into main memory.

*addr* specifies the starting address, while *size* specifies the amount of MS-DOS memory, which will be loaded with the contents of EMS memory. The size of the area is checked, i.e. the area of memory read in with EMEMPOP must be exactly the same as the area saved with EMEMPUSH.

**Example:**

```
DIM a(100,100)
FOR i%=1 TO 100
  FOR j%=1 TO 100
    a(i%,j%)=j%
  NEXT j%
NEXT i%
EMEMPUSH V:a(0,0),101 << 3,"1.Column"
ERASE a()
DIM b(100)
EMEMPOP V:b(0),101 << 3
FOR i%=1 TO 100
  PRINT b(i%)'
NEXT i%
```

## Commands and functions

---

```
// Dimensions the floating point array a() and fills
// it with numbers from 0 to 100, column-wise. The
// first column in a() is then moved to EMS and a()
// is deleted. Array b() is then dimensioned and the
// values in first column of former array a() are
// moved to b().
```

### Remarks:

-

addr: see the {} function.

### See

### Also:

EPUSH, EPOP, EGET, EKILL, EDIR, EAVAIL,  
EPARLOAD, EPARSAVE, EMEMPUSH,  
EMEMGET

## EMEMPUSH Command

**Action:** saves contents of MS-DOS RAM to Expanded Memory (EMS).

**Syntax:** EMEMPUSH addr,size["NAME"]  
*addr: address*  
*size: iexp*

**Abbreviation:** -

**Explanation:** EMEMPUSH addr,size["name"] works like EPUSH, except that, instead of variables and arrays, whole areas of MS-DOS RAM are saved to EMS memory. This enables, for example, saving of portions of an array stored to EMS memory. It is recommended to give this area a name for subsequent reading with EMEMGET.

addr specifies the starting address, while size specifies the amount of memory to save.

**Example:**

```
DIM a(100,100)
FOR i%=1 TO 100
  FOR j%=1 TO 100
    a(i%,j%)=j%
  NEXT j%
NEXT i%
EMEMPUSH V:a(0,0),101 << 3,"1.Column"
ERASE a()
DIM b(50)
EMEMGET V:b(0),51 << 3,"1.Column"
FOR i%=1 TO 100
  PRINT b(i%)'
NEXT i%
```

## Commands and functions

---

```
// Dimensions the floating point array a() and fills
// it with numbers from 0 to 100, column-wise. The
// first column in a() is then moved to EMS and a()
// is deleted. Array b() is then dimensioned and the
// first 51 values in first column of former array
// a() are moved to b().
```

### Remarks:

-

addr: see the {} function.

### See

### Also:

EPUSH, EPOP, EGET, EKILL, EDIR, EAVAIL,  
EPARLOAD, EPARSAVE, EMEMPOP,  
EMEMGET

## END Command

**Action:** terminates a GFA-BASIC program.

**Syntax:** END

**Abbreviation:** -

**Explanation:** END terminates a GFA-BASIC program. It causes an alert box with "Program end" and "Return" to appear.

**Example:**

```
CLS
SCREEN 16           // EGA mode
OPENW #1,10,10,400,200,-1
FOR i%=1 TO 10
    PRINT i%
NEXT i%
~INTR(&16,_AH=0)
END
```

```
// Turns the graphic mode on and opens a window. The
// digits from 1 to 10 are then written in the window
// and the program waits for a keypress. If the key
// is pressed the text mode is turned back on and the
// message "Program end" is displayed. When the
// Return key is pressed the GFA-BASIC editor is
// invoked.
```

**Remarks:** -

**See**

**Also:** EDIT

### EOF() Function

**Action:** tests if the data pointer points to the end of a file.

**Syntax:** EOF(#n)  
*n: iexp*

**Explanation:** EOF(#n) always acts on the file on the previously opened channel *n*, and returns -1 if the data pointer points to the end of this file or 0 if not.

**Example:**

```
OPEN "i",#1, "TEST.TXT"
DO UNTIL EOF(#1)
  INPUT #1,a$
  PRINT " ";a$
LOOP
CLOSE #1

// Opens the file TEST.TXT in current directory and
// reads its contents until file end.
```

**Remarks:** -

**See**

**Also:** LOC(), LOF()

## EPARLOAD Command

**Action:** data exchange between different GFA-BASIC programs via Expanded Memory (EMS)

**Syntax:** EPARLOAD "File Name"  
*File Name:* *sexp*

**Abbreviation:** -

**Explanation:** In normal circumstances, when GFA-BASIC interpreter ends, the EMS memory is erased. However, there are cases when it would make sense to preserve the contents of variables which were saved to EMS memory for subsequent use by another program.

In such a case, the current configuration of EMS memory can be saved to a file by using EPARSAVE and loaded back from within a GFA-BASIC program by using EPARLOAD. "File Name" specifies the file which will contain the relevant EMS configuration. To avoid disruption of EMS configuration EPARSAVE should always be performed at the end, and EPARLOAD at the beginning of a GFA-BASIC program.

**Beispiel:** saves the current EMS configuration

```
FOR i% 1 10          // describe EMS
  EPUSH i%
NEXT i%
EPARSAVE "1T010.EMS" // save EMS configuration
```

## Commands and functions

---

```
// get EMS configuration:
EPARLOAD "1T010.EMS" // load EMS configuration
FOR i% 1 10
  EPOP j%
  PRINT j%'
NEXT i%
```

**Bemerkung:** If other programs change the EMS memory pages used by GFA-BASIC after an EPARSAVE, EPARLOAD should not be used.

**See**

**Also:**

EPARSAVE



## EPARSAVE Command

**Action:** data exchange between different GFA-BASIC programs via Expanded Memory (EMS)

**Syntax:** EPARSAVE "File Name"  
*File Name:* *sexp*

**Abbreviation:** -

**Explanation:** In normal circumstances, when GFA-BASIC interpreter ends, the EMS memory is erased. However, there are cases when it would make sense to preserve the contents of variables which were saved to EMS memory for subsequent use by another program.

In such a case, the current configuration of EMS memory can be saved to a file by using EPARSAVE and loaded back from within a GFA-BASIC program by using EPARLOAD. "File Name" specifies the file which will contain the relevant EMS configuration. To avoid disruption of EMS configuration EPARSAVE should always be performed at the end, and EPARLOAD at the beginning of a GFA-BASIC program.

**Example:** see the EPARLOAD function

**Remarks:** see the EPARLOAD function.

**See Also:** EPARLOAD

# EPOP Command

**Action:** reads variables from Expanded Memory (EMS).

**Syntax:** EPOP a,a(),...  
*a:* variable of any type (a%,a&,...)  
*a():* array of any variable type (a(), a\$( ),...)

**Abbreviation:** -

**Explanation:** EPOP a\$,a(),... reads the list of variables and/or arrays from EMS memory into MS-DOS RAM. The EMS memory used by EPUSH and EPOP is organized on "First In Last Out" principle, i.e. as a stack. EPOP gets access to the last variable saved with EPUSH, moves it to main memory and deletes it from EMS. It is, thereby, irrelevant whether the variable saved with EPUSH has the same name as the variable read with EPOP.

Both variables must be of the same variable type. In case of arrays, an internal dimensioning is performed before accessing the data. When the variables and/or arrays saved in EMS are retrieved back, a type check is performed.

Do note, that when specifying a list of variables with EPOP, these variables will be read in in the reverse order. This makes it possible to use the same variable lists for EPUSH and EPOP commands, in spite of the "First In Last Out" principle.

**Example:**

```
SCREEN 3
n%=100
DIM a(n%,1)
//
FOR i%=0 TO n%
  a(i%,0)=i%
```

```
a(i%,1)=n%+i%
NEXT i%
//
EPUSH a(): "ARRAY 1"
DIM a(n%,1)           // does not result in an
//                   error, since the first
//                   array is now in EMS and an
//                   ERASE a() was performed on
//                   it.
//
ERASE a()
//
EPOP a()              // does not result in an
//                   error since EPOP performs
//                   an internal DIM a().
//
FOR i%=0 TO n%
  PRINT a(i%,0)'
  PRINT a(i%,1)'
NEXT i%

// Creates a twodimensional floating point array and
// fills it with numbers from 0 to n% and from n% to
// 2*n%. The array is then moved to EMS.
// An array with the same name a() is again
// dimensioned and deleted. The first array is then
// read back from the EMS memory printed out.
```

**Remarks:**

-

**See**

**Also:**

EPUSH, EGET, EDIR, EAVAIL, EKILL,  
EPARLOAD, EPARSAVE, EMEMPUSH,  
EMEMPOP, EMEMGET

# EPUSH Command

**Action:** writes variables into Expanded Memory (EMS).

**Syntax:** EPUSH a[:"Name"],a()[:"Name"],...  
*a:* variable of any type (*a\$, a#, a%,...*)  
*a():* array of any variable type  
(*a\$( ), a#( ),...*)  
*"Name"* the name of *a* or *a()*

**Abbreviation:** -

**Explanation:** EPUSH a[:"Name"],a()[:"Name"],... writes the variables and/or arrays to EMS memory. Each variable or array can, optionally, be followed by a colon!, and a character string up to 16 characters long, which names this variable or array. This name serves to identify the variable or array for access to EMS memory with EGET (see EGET). The variables and arrays themselves must be separated by commas.

Even though it only makes sense to store in EMS the variables which require a lot of memory, in principle, any variable can be saved to EMS.

When an array is saved to EMS, EPUSH performs an ERASE on this array in the MS-DOS RAM. This means that after EPUSH, the corresponding array does not exist in main memory anymore.

**Example:**

```
SCREEN 3
n%=100
DIM a(n%,1)
//
FOR i%=0 TO n%
  a(i%,0)=i%
  a(i%,1)=n%+i%
NEXT i%
```

```
//
EPUSH a(): "ARRAY 1"
DIM a(n%,1)           // does not result in an
//                   error, since the first
//                   array is now in EMS and an
//                   ERASE a() was performed on
//                   it.

//
ERASE a()
//
EPOP a()              // does not result in an
//                   error since EPOP performs
//                   an internal DIM a().
//
//
FOR i%=0 TO n%
    PRINT a(i%,0)'
    PRINT a(i%,1)'
NEXT i%

// Creates a twodimensional floating point array and
// fills it with numbers from 0 to n% and from n% to
// 2*n%. The array is then moved to EMS.
// An array with the same name a() is again
// dimensioned and deleted. The first array is then
// read back from the EMS memory printed out.
```

**Remarks:**

-  
.

**See**

**Also:**

EPOP, EGET, EDIR, EAVAIL, EKILL,  
EPARLOAD, EPARSAVE, EMEMPUSH,  
EMEMPOP, EMEMGET

## EQV() Function

**Action:** returns the bit-wise equivalent of two bit patterns.

**Syntax:** EQV(*i,j*)  
*i,j*: *exp*

**Explanation:** EQV(*i,j*) sets in the result only the bits which are the same in both *i* and *j*. This function is equivalent to NOT(XOR(*i,j*)).

**Example:**

```
PRINT BIN$(3,4)           // prints 0011
PRINT BIN$(10,4)          // prints 1010
PRINT BIN$(XOR(3,10),4)   // prints 1001
PRINT BIN$(EQV(3,10),4)   // prints 0110
PRINT BIN$(NOT(XOR(3,10)),4) // prints 0110
```

```
// EQV (3,10) returns the value -10. This result is
// easier to understand when all 32 bits are shown:
```

```
BIN$(3,32)      = 00000000000000000000000000000011
BIN$(10,32)     = 000000000000000000000000000001010
BIN$(EQV(3,10),32)= 111111111111111111111111111110110
```

**Remarks:** -

**See**

**Also:** AND(), OR(), XOR(), IMP()

## ERASE Command

**Action:** deletes all arrays listed after it.

**Syntax:** ERASE x1()[,x2(),...]  
*x1(),x2(),...: arrays of any type*

**Abbreviation:** era x1(),...

**Explanation:** The arrays in the list after ERASE must be separated by commas.

**Example:**

```
DIM x(10),y(20),n%(17,3),a$(23)
ERASE x(),n%()
PRINT n%(2,1)           // reports an error
```

**Remarks:** -

**See**

**Also:** CLR

### ERR Variable

**Action:** error code number

**Syntax:** -

**Explanation:** When an error occurs, the ERR variable contains its code number (see the list of error messages in the appendix).

**Example:**

```
INPUT "Which error should be shown";a%
ERROR a%
PRINT "This was the error number ";ERR
```

**Remarks:** -

**See**

**Also:** ON ERROR, ERROR, ERR\$(), FATAL



## ERR\$( ) Function

**Action:** determines an error.

**Syntax:** ERR\$(i)  
*i:* *iexp*

**Explanation:** The ERR\$(i) function returns the string containing the GFA-BASIC error message for code number i (see the list of error messages in the appendix).

**Example:**

```
FOR i%=1 TO 10
  PRINT ERR$(i%)
NEXT i%
```

// Returns the strings with GFA-BASIC error messages  
// for codes 1 to 10.

**Remarks:** -

**See**

**Also:** ON ERROR, ERROR, ERR, FATAL

### ERROR Command

**Action:** triggers an error.

**Syntax:** ERROR *n*  
*n*: *iexp*

**Abbreviation:** -

**Explanation:** ERROR *n* displays an error with the number *n* (see the list of error messages in the appendix).

**Example:** INPUT "Which error should be shown";a%  
ERROR a%

**Remarks:** -

**See**

**Also:** ON ERROR, ERR, ERR\$(), FATAL

## EVEN() Numeric function

**Action:** tests if a numeric expression is even and returns -1 (true) if it is, or 0 if the expression is odd.

**Syntax:** EVEN(x)  
*x:* *aexp*

**Example:**

```
x=2
PRINT EVEN(x*3)      // prints  -1
PRINT EVEN(x*3+1)    // prints   0
```

**Remarks:** -

**See**

**Also:** ODD()

### EXEC Command

**Action:** loads and runs a program from within the GFA-BASIC interpreter.

**Syntax:** EXEC *nam,cmdl*  
*nam, cmdl: sexp*

**Abbreviation:** -

**Explanation:** The EXEC *nam,cmdl* command serves to load and run a program from within the GFA-BASIC interpreter. The invoked program returns, after terminating, to the GFA-BASIC interpreter.

The *nam* expression contains the name of the program to load (and run). The program name includes the full pathname.

The *cmdl* expression contains the command line which is inserted in the program segment prefix of the called program.

**Example:**

```
CHDRIVE "C:"  
IF EXIST ("COMMAND.COM")  
    EXEC COMMAND.COM,""  
ELSE  
    PRINT "Program not found"  
ENDIF
```

**Remarks:** -

**See**  
**Also:** -

## EXEC() Function

**Action:** returns a value from the invoked program.

**Syntax:** EXEC(nam,cmdl)  
*nam,cmdl: sexp*

**Explanation:** The EXEC(nam,cmdl) function invokes program nam and gives it the command line cmdl.

The nam expression contains the name of the called program. The program name includes the full path-name.

The cmdl expression contains the command line which is inserted in the program segment prefix of the called program.

**Example:**

```
CHDRIVE "C:"  
a%=EXEC("COMMAND.COM","")  
PRINT a%
```

// Returns the value from the program COMMAND.COM.

**Remarks:** -

**See**

**Also:** -

### EXIST Function

**Action:** determines if a particular file exists.

**Syntax:** EXIST(a\$)  
*a\$: sexp; path name of a file*

**Explanation:** The EXIST(a\$) function determines if a particular file exists in the path name specified in a\$. EXIST() returns -1 if this file exists or 0 if not.

**Example:**

```
a$="D:\TEST\A1\  
IF EXIST(a$+"HELP.TXT")  
    OPEN "u","D:\TEST\A1\HELP.TXT", #1  
    PRINT #1,"TEST...TEST...TEST"  
    CLOSE #1  
ENDIF  
  
// Tests if file HELP.TXT exists in directory  
// \TEST\A1 on drive D and writes, if it does prints  
// the string TEST...TEST...TEST.
```

**Remarks:** -

**See**

**Also:** -

## EXIT IF Structure

**Action:** serves to terminate a loop when the condition following EXIT...IF is logically "true".

**Syntax:** EXIT IF condition  
*Condition: any numerical, logical or string condition*

**Abbreviation:** ex condition

**Explanation:** The EXIT.IF command makes it possible to test and exit any loop for a condition other than the one specified in the loop itself (see FOR...NEXT, WHILE...WEND, REPEAT...UNTIL and DO...LOOP). In contrast to the GOTO command, a loop is terminated in an "orderly" fashion by using EXIT IF.

In other words, EXIT IF always jumps to the first programming statement after the last line of the loop, while GOTO can jump anywhere within a program, PROCEDURE or FUNCTION.

**Example:**

```
a%=1
FOR i%=1 TO n%
  a%*=i%
  EXIT IF a%>32000
NEXT i%
```

```
// Calculates the factorial of n and stores the
// result in the variable a%. The calculation is
// terminated if the result exceeds 32000.
```

## Commands and functions

---

**Remarks:** The IF condition THEN EXIT DO (or LOOP) command common to other dialects of BASIC can also be used and will be converted by GFA-BASIC into EXIT IF condition // DO (or LOOP).

**See**  
**Also:** GOTO



## EXP() Numeric function

**Action:** returns the EULER's number  $e$  ( $= 2.178\dots$ ) to the power of a numeric expression.

**Syntax:** EXP( $x$ )  
 $x$ : *aexp*

**Explanation:** EXP( $x$ ) calculates the  $x$ -th power of EULER's number  $e = 2.178\dots$

**Example:** PRINT EXP(SQR(2))      // prints      4.11325...

**Remarks:** EXP( $x$ ) is the reverse function of LOG( $x$ ), which means:

EXP(LOG(PI)) = PI = 3.14...

**See**

**Also:** LOG()

### FACT() Numeric function

**Action:** returns the factorial of a natural number.

**Syntax:** FACT(*n*)  
*n*: *exp*

**Explanation:** FACT(*n*) returns the factorial of a natural number *n* (*n*!). A factorial is the product of the first *n* natural numbers, where  $0! = 1$ .

**Example:** PRINT FACT(6) // prints 720

**Remarks:** -

**See**  
**Also:** COMBIN(), VARIAT()

## FALSE Variable

**Action:** 0 constant for logical false

**Syntax:** a!=FALSE

**Explanation:** The result of comparison 0 < > is FALSE=0.

**Example:**

```
i%=20
IF i%
    a!=TRUE
    PRINT "i% is not equal to 0; a!=";a!
ENDIF
i%=0
IF !i%
    a!=FALSE
    PRINT "i% is equal to 0; a!=";a!
ENDIF

// prints
// i% is not equal to 0; a!=-1
// i% is equal to 0; a!=0
```

**Remarks:** -

**See**

**Also:** TRUE

### FATAL Variable

**Action:** evaluates an error.

**Syntax:** -

**Explanation:** The variable **FATAL** is true when an error produces an address which is unknown to the interpreter. This can occur when, for example, an operating system routine is invoked. If **FATAL** is true, a **RESUME** or **RESUME NEXT** cannot be executed correctly.

**Example:** better not !

**Remarks:** -

**See**

**Also:** ON ERROR, ERROR, ERR, ERR\$()

## FGETDTA Function

**Action:** returns the address of the disk transfer area (DTA).

**Syntax:** `addr=FGETDTA()`  
*addr: addresss*

**Explanation:** see FSETDTA

**Example:**

```
//Example FGETDTA FSFIRST FSNEXT
//
TYPE dta:
-CHAR *21                fs_donttouch$
-BYTE                    fs_attr
-CARD                    fs_time
-CARD                    fs_date
-LONG                    fs_size
-CHAR *14                fs_name$
ENDTYPE
//
dta%=FGETDTA()           // stores the address of
//                        DTA in a%
//
e%=FSFIRST("*.\"",%10001)
WHILE e%=>0
  a$=SPACE$(60)
  MID$(a$,2)={dta%}.fs_name$
  q$=" "
  IF BTST({dta%}.fs_attr,4)
    RSET q$="<DIR>"
  ELSE
    RSET q$=STR$({dta%}.fs_size)
  ENDIF
  MID$(a$,16)=q$
  a%={dta%}.fs_date
  q$=DEC$(a% & 31,2)+". "
  q$=q$+DEC$((a% >> 5) & 15,2)+". "
```

## Commands and functions 3-499

---

```
q$=q$+DEC$((a% >> 9) + 1980,4)
MID$(a$,30)=q$
a%={dta%}.fs_time
q$=DEC$(a% >> 11,2)+". "+DEC$((a% >> 5) & 63,2)
q$=q$+DEC$((a% & 31)*2,2)
MID$(a$,42)=q$
PRINT a$
REPEAT
UNTIL LEN(INKEY$)
e%=FSNEXT()
WEND
```

```
// Reads all entries which are not hidden from the
// current directory, including all subdirectories,
// and waits for a keypress after displaying each
// entry. This continues until all entries are shown.
```

**Remarks:** addr: see the {} function.

**See**

**Also:** FSETDTA(), FSFIRST(), FSNEXT()

## FIELD # ...AS...AT Command

**Action:** random access file management

**Syntax:** FIELD #n,count AS set\$[,count AS set\$,...] or  
FIELD #n,count AT(x)[,count AT(x),...] or  
a combination of AS and AT

*n:* *iexp; channel number*  
*count:* *iexp*  
*set\$:* *svar, but not an array variable*  
*x:* *addr*

**Explanation:** RANDOM ACCESS files are composed of records and fields. A record is a collection of data, for example an address. A record contains a mixture of fields (the record Address can, for example, be divided into fields: Name, Street, Postcode and City). Both records and fields have a set size.

FIELD AS divides records into fields. n is the channel number (from 0 to 99) of a file previously opened with OPEN. The integer count defines the corresponding field length. The string variable set\$ always refers to one field in a record. If a record is divided into several fields, each must be separated with a comma (count AS set\$). The sum of individual field sizes must equal the length of the record. To save individual fields with length given in count, the commands LSET, RSET and MID\$ should be used. Using FIELD AT numerical variables can be written to an R-file (random access) without having to convert them to strings. The pointer to numerical variables which are to be saved is given in brackets after AT and the number of bytes to read from this address is given before AT. A mixture of AS and AT is allowed.

## Commands and functions

---

The FIELD command can span across several program lines. The maximum record size is 32767 bytes with the maximum number of fields of about 5000.

**Example:**

```
OPEN "R",#1,"A:\Addresses.DAT",62
FIELD #1,24 AS name$,24 AS street$ AT(*postcode&),12
FIELD #1,2 AS city$
.
.
CLOSE #1
```

**Remarks:**

addr: see the {} function.

**See**

**Also:**

GET, PUT, RECORD



## FILES...TO Command

**Action:** prints the directories in the current path name.

**Syntax:** FILES path\$ [TO file\$]  
*path\$:* *sexp; current path name*  
*file\$:* *sexp; optional file name*

**Abbreviation:** fil ...

**Explanation:** DIR path\$ returns the directories in path name specified in path\$. If path\$ ends with a ":" or "\", GFA-BASIC automatically appends ".\*". The default destination for the output of the directory is to screen.

In contrast to DIR ... TO each file in FILES ... TO is followed by date, time and file size.

The specification of TO file\$ is optional. It can be used to divert the directory output to a file or a peripheral device.

**Example:**

```
path$="A:\TEST\A1:"
FILES path$
//
// prints the directory of A:\TEST\A1 to screen
//
FILES path$ TO "A:\VERZ.TXT"

// writes the directory of A:\TEST\A1 to file
// A:\VERZ.TXT
//
FILES path$ TO "PRT:"
//
// print the directory of A:\TEST\A1 to printer
//
FILES path$+"*.TXT" TO "PRN"
//
```

## Commands and functions

---

```
// prints all files with extension TXT from  
// A:\TEST\A1 to printer
```

**Remarks:**

-

**See**

**Also:**

DIR ... TO

## FILESELECT Command

**Action:** file selection

**Syntax:** FILESELECT path\$,default\$,name\$  
*path\$,default\$:* *sexp*  
*name\$:* *svar*

**Abbreviation:** filesel path\$,default\$,name\$

**Explanation:** The FILESELECT pfad\$,default\$,name\$ command invokes the File-Selector-Box, which lists the directory specified in path\$. A file can be selected from this directory by using the mouse or by typing in the file name.

The path\$ expression contains the name of the drive and the directory which should be listed. If no path is specified the current directory is shown.

default\$ contains the name of a file, which is appears in the File-Selector-Box by default.

After a file has been selected its name is returned in name\$. If the Cancel button is clicked instead name\$ contains a blank string.

**Example:**

```
FILESELECT "E:\GFABASIC\","",n$
IF LEN(n$)
  IF RIGHT$(n$)="\"
    PRINT "You haven't selected a file"
  ELSE
    PRINT "The file ";n$" was selected"
  ENDIF
ELSE
  PRINT "The Cancel button was clicked"
ENDIF
```

**Remarks:**

-

**See**

**Also:**

-

## FILL Command

**Action:** fills any enclosed area.

**Syntax:** FILL *x,y*[,*f*]  
*x,y,f*: *iexp*

**Abbreviation:** fi *x,y*[,*f*]

**Explanation:** FILL *x,y*[,*f*] fills an enclosed area. The fill starts at the coordinates *x,y*. If the optional parameter *f* is specified the fill is limited to the points with colour *f*.

**Example:**

```
LINE 0,280,639,280
FOR i%=1 TO 10
  BOX MUL(i%,20),200,SUB(ADD(MUL(i%,20),20),i%),280
  TEXT SUB(MUL(i%,20),4),295,i%
  DEFFILL ,2,i%
  FILL SUCC(MUL(i%,20)),201,1
NEXT i%

// Draws rectangles with various patterns repeatedly
// on a straight line and performs a fill limited to
// one colour.
```

**Remarks:** -

**See  
Also:** -

### FIX() Function

**Action:** returns the integer portion of a numeric expression.

**Syntax:** `FIX(x)`  
*x: aexp*

**Explanation:** FIX removes the fractional portion, i.e. it rounds the number down.

**Example:**

```
PRINT FIX(3*1.2)    // prints    3
PRINT FIX(3*-1.2)   // prints   -3
PRINT TRUNC(3*1.2)  // prints    3
PRINT TRUNC(3*-1.2) // prints   -3
```

**Remarks:** TRUNC() is synonymous with FIX() and can be used instead.

**See**

**Also:** FLOOR(), INT(), CEIL(), TRUNC(), FRAC()

## FLOOR() Function

**Action:** returns the next smaller integer number.

**Syntax:** FLOOR(x)  
*x:* *aexp*

**Explanation:** FLOOR() is equivalent to rounding off towards  $-\infty$ .

**Example:**

```
PRINT FLOOR(3*1.2)    // prints    3
PRINT FLOOR(-3*1.2)   // prints   -4
PRINT INT(3*1.2)      // prints    3
PRINT INT(-3*1.2)     // prints   -4
```

**Remarks:** INT() is synonymous with FLOOR() and can be used instead.

**See**

**Also:** INT(), CEIL(), TRUNC(), FIX(), FRAC()

### FN Command

**Action:** unconditional branch.

**Syntax:** FN functionname[(parameterlist)]  
*functionname:* the name of the function declared using FUNCTION or DEFFN

**Abbreviation:** @

**Explanation:** The FN functionname command causes an unconditional branch to the FUNCTION with the specified name.

If the function has not been declared correctly (for example invalid name or parameter list) an error is reported.

@functionname can be used instead of FN functionname".

**Example:**

```
DATA 12,3.4,5,6.7,8,9,12,14
DIM a(7)
sq=0
FOR i%=0 to 7
    READ a(i%)
    ADD sq,a(i%)^2
NEXT i%
value=DIV(sq-8*FN am(7,a()),7)
PRINT "The standard deviation is: ";value
END
FUNCTION am(n% VAR vector())
    LOCAL i%,s
    IF n%<0
        RETURN error_code
    ELSE IF n%=0
        RETURN vector(0)
    ENDIF
```



```
FOR i%=0 TO n%
  ADD s,vector(i%)
NEXT i%
RETURN DIV(s,SUCC(n%))
ENDFUNC
```

```
// In this example a function is called on the
// "value=..." line, and the value returned from the
// function is used as a part of an arithmetic
// expression.
```

```
DEFFN nth_root(x,n%)=x^1/n%
PRINT "3rd root of PI =" ; FN nth_root(PI,3)
```

```
// In this example FN is used to jump to a one-line
// nth_root function.
```

**Remarks:**

-

**See**

**Also:**

GOSUB, ON...GOSUB, ON...condition...GOSUB, @,  
GOTO

### FORM INPUT Command

**Action:** entry of string variables

**Syntax:** FORM INPUT *n*,*a\$*  
*n*: *iexp*  
*a\$*: *svar*

**Abbreviation:** -

**Explanation:** FORM INPUT is used to enter individual string variables. The integer expression *n* defines the maximum length of *a\$* (from 1 to 255). For further explanation see INPUT bearing in mind, however, that FORM INPUT can only accept one single string variable.

**Example:** FORM INPUT 10,*a\$*

```
// Waits for input of a character string up to 10
// characters in length (including spaces and
// CHR$(0)0.
```

**Remarks:** see the INPUT function.

**See**

**Also:** INPUT, LINE INPUT, INPUT #, LINE INPUT #,  
FORM INPUT AS

## FORM INPUT AS Command

**Action:** entry of string variables

**Syntax:** FORM INPUT *n* AS *a\$*  
*n*: *iexp*  
*a\$*: *svar*

**Abbreviation:** -

**Explanation:** FORM INPUT is used to enter individual string variables. The integer expression *n* defines the maximum length of *a\$* (from 1 to 255). In contrast to FORM INPUT, the FORM INPUT AS command displays the current contents of *a\$*, which can be edited as described in INPUT.

For further explanation see INPUT bearing in mind, however, that FORM INPUT can only accept one single string variable.

**Example:** *a\$*="Hello GFA"  
FORM INPUT 10 AS *a\$*

```
// Prints Hello GFA and waits for input of a string  
// of up to 10 characters in length (including spaces  
// and CHR$(0).
```

**Remarks:** see the INPUT function.

**See**

**Also:** INPUT, LINE INPUT, FORM INPUT, INPUT #,  
LINE INPUT #

### FOR...NEXT Structure

**Action:** a programming loop which is executed the specified number of times.

**Syntax:**

```
FOR i=x TO y [STEP z]
    // programsegment
NEXT i
```

*i: avar; any numeric variable*  
*x,y,z: aexp; arithmetic expression*

**Abbreviation:**

```
f i x y [z]
    // programsegment
n i
```

**Explanation:** Each FOR...NEXT loop begins by initialising the loop counter *i* to the specified starting value. With each run the loop counter is incremented or decremented by the specified amount (in case of default by 1, otherwise by the step value in *z*). When the counter over- or underflows the loop criterion in *y*, the command after the next NEXT is unconditionally branched to.

In the following structure:

```
FOR i=x TO y
    // programsegment
NEXT i
```

the loop counter *i* is incremented by 1 every time the loop runs through NEXT *i*. The loop ends when *i* overflows the loop criterion value *y*.

In the following structure:

```
FOR i=x TO y STEP z
    // program segment
NEXT i
```

Every time the loop runs through NEXT.i, the loop counter i is incremented by step amount in z, if this amount is positive, or is decremented by step amount in z, if this amount is negative.

The loop ends when i, for  $\text{SGN}(z) = 1$ , overflows the loop criterion value y or, for  $\text{SGN}(z) = -1$ , underflows the loop criterion value y.

The following structure:

```
FOR i=x DOWNT0 y
    // programsegment
NEXT i
```

is a special case of:

```
FOR i=x TO y STEP z,
when z=-1.
```

The loop counter i is decremented by 1 every time the loop runs through NEXT i. The loop ends when i underflows the loop criterion value in y.

If, at the very beginning of the loop, the loop counter i is already greater than (for FOR...TO) or less than (for FOR...DOWNT0 or FOR...TO...STEP z, when  $z < 0$  the loop criterion y, the loop is not executed.

In the following structure:

```
FOR i=x TO i+y
    // programsegment
NEXT i
```

It goes from the value it has at loop start until the value in the expression (i+y) after TO.

By using an EXIT IF command, the FOR...NEXT loop can be terminated regardless of whether the loop condition is fulfilled.

## Commands and functions

---

Do note, that the loop criterion in the FOR...NEXT loop must always be numeric!

For loop criteria which are not numeric the DO...WHILE, REPEAT...UNTIL or DO...LOOP loops must be used.

### Example:

```
FOR i&=1 to 10
  PRINT i&
NEXT i&                                // prints the numbers from 1
//                                    to 10 on the screen.
```

```
FOR i=0 TO 1 STEP 0.1
  .. PRINT i
NEXT i                                // prints the numbers
//                                0,0.1,0.2,0.3 ... 1 on
//                                the screen.
```

```
FOR i=1 to 0 STEP -0.1
  PRINT i
NEXT i                                // prints the numbers
//                                1,0.9,0.8,0.7,...,0 on
//                                the screen.
```

```
FOR i&=10 DOWNT0 1
  PRINT i&
NEXT i&                                // prints the numbers from 10
//                                to 1 (decreasing) on the
//                                screen.
```

```
i&=6
FOR i&=1 to i&*9
  PRINT i&
NEXT i&           // prints the numbers from 1
//               to 42 on the screen.
```

**Remarks:** As long as different loop counters are used the FOR...NEXT loops can be embedded to any number of levels.

**See**

**Also:** WHILE...WEND, REPEAT...UNTIL, DO...LOOP

### FRAC() Numeric function

**Action:** returns the fractional part of a numeric expression.

**Syntax:** `FRAC(x)`  
*x:* *aexp*

**Explanation:** The value returned from `FRAC(x)` has the same sign as *x*.

**Example:**

```
PRINT FRAC(3*1.2)    // prints    6
PRINT FRAC(3*-1.2)   // prints   -6
```

**Remarks:**  $x = \text{TRUNC}(x) + \text{FRAC}(x)$

**See**

**Also:** `FLOOR()`, `INT()`, `CEIL()`, `TRUNC()`, `FIX()`



## FRE() Function

**Action:** reports the amount of free (main) memory.

**Syntax:** FRE() or  
FRE(*x*)  
*x*: *iexp*

**Explanation:** The FRE() function calculates the amount of free memory for strings available without an additional request for another 64 K block. The dummy parameter FRE(0) performs a garbage collection in the string area first. Without the parameter the free memory is returned without the garbage collection. The total amount of free memory can be obtained with MALLOC(-1)

**Example:** PRINT FRE()  
  
// Returns the amount of free memory.

**Remarks:** -

**See**

**Also:** EAVAIL

### FREEFONT Command

**Action:** deletes one of the externally loaded fonts

**Syntax:** FREEFONT

**Abbreviation:** freef

**Explanation:** FREEFONT deletes from memory the character set loaded with LOADFONT p\$. If the new character is not loaded with another LOADFONT, the previously active system character set is used.

**Example:** FREEFONT

**Remarks:** -

**See**

**Also:** LOADFONT

## FSETDTA Command

**Action:** sets the address for the disk transfer area (normally at `_PSP + $80`).

**Syntax:** `FSETDTA(addr)`  
*addr: address*

**Explanation:** The functions \$4E (FSFIRST=get first file entry) and \$4F (FSNEXT=get next file entry) store the data about found files in the DTA. The first 42 bytes of the DTA have the following layout:

OFFSET	Length	Contents
+\$00	21 bytes	reserved
+\$15	1 byte	file attributes
+\$16	2 bytes	time of last modification
+\$18	2 bytes	date of last modification
+\$1A	4 bytes	file size
+\$1E	12 bytes	file name and extension only, no pathname

The layout of the attribute byte is as follows

bit	meaning when set
0	write protected file
1	hidden file
2	system file
3	volume label
4	directory
5	archive bit

**Example:** `FSETDTA(_PSP+$80)`

`// Sets the DTA to the standard address`

**Remarks:** `addr`: see the `{ }` function.

## Commands and functions

---

**See**

**Also:**

BASEPAGE, \_PSP, FGETDTA, FSFIRST, FSNEXT

## FSFIRST() Function

**Action:** searches for the first file that satisfies the given criteria

**Syntax:** `m = FSFIRST(q$,n)`  
*m:* 16 bit integer variable  
*q\$* sexp; search criteria  
*n:* iexp; attributes

**Explanation:** see FSETDTA

**Example:**

```
// Example FGETDTA FSFIRST FSNEXT
//
TYPE dta:
-CHAR *21                fs_donttouch$
-BYTE                    fs_attr
-CARD                    fs_time
-CARD                    fs_date
-LONG                    fs_size
-CHAR *14                fs_name$
ENDTYPE
//
dta%=FGETDTA()           // stores the address
//                        DTA in a%
//
e%=FSFIRST("*. **",%10001)
WHILE e%=>0
  a$=SPACE$(60)
  MID$(a$,2)={dta%}.fs_name$
  q$="          "
  IF BTST({dta%}.fs_attr,4)
    RSET q$="<DIR>"
  ELSE
    RSET q$=STR$({dta%}.fs_size)
  ENDIF
  MID$(a$,16)=q$
  a%={dta%}.fs_date
```

## Commands and functions

---

```
q$=DEC$(a% & 31,2)+". "  
q$=q$+DEC$((a% >> 5) & 15,2)+". "  
q$=q$+DEC$((a% >> 9) + 1980,4)  
MID$(a$,30)=q$  
a%={dta%}.fs_time  
q$=DEC$(a% >> 11,2)+". "+DEC$((a% >> 5) & 63,2)  
q$=q$+DEC$((a% & 31)*2,2)  
MID$(a$,42)=q$  
PRINT a$  
REPEAT  
UNTIL LEN(INKEY$)  
e%=FSNEXT()  
WEND
```

```
// Reads all entries which are not hidden from the  
// current directory, including all subdirectories,  
// and waits for a keypress after displaying each  
// entry. This continues until all entries are shown.
```

**Remarks:**

-

**See**

**Also:**

FSETDTA(), FGETDTA(), FSNEXT()

## FSNEXT() Function

**Action:** searches for the next file according to the criteria set with FSFIRST.

**Syntax:** `m = FSFIRST()`  
*m:* *iexp*

**Explanation:** see FSETDTA

**Example:** see FSFIRST()

**Remarks:** -

**See**

**Also:** FSETDTA(), FGETDTA(), FSFIRST()

### FULLW# Graphic command

**Action:** expands a window to its maximum size

**Syntax:** FULLW #n  
*n: iexp*

**Abbreviation:** -

**Explanation:** Expands a window to full screen size or opens such a window.

**Example:**

```
SCREEN 16 // EGA mode
OPENW #1,10,10,200,100,-1
REPEAT
  IF MOUSEK AND 1
    FULLW #1
  ENDIF
  a$=inkey$
UNTIL a$=chr$(27)
CLOSEW #1
SCREEN 3
EDIT
```

```
// Draws a window on the screen and expands it to its
// maximum size when the left mouse button is
// clicked.
```

**Remarks:** -

**See**

**Also:** OPENW, CLOSEW, TITLEW, INFOW, TOPW, SIZEW, CLEARW



## FUNCTION...name...ENDFUNC Structure

**Action:** evaluates an arithmetic expression which is repeatedly used throughout the program, whereby the result of the expression changes depending on the variables passed to it.

**Syntax:**

```
FUNCTION name [(v1,...vn,VAR arr1(),...arrn(),
               r1,...,rn)]
    LOCAL l1,...ln
    // programsegment
    RETURN aexp
ENDFUNC
```

<i>name:</i>	<i>function name</i>
<i>v1,...,vn:</i>	<i>'Call by Value' variables</i>
<i>arr1(),...,arrn():</i>	<i>arrays</i>
<i>r1,...,rn:</i>	<i>'Call by Reference' variables</i>
<i>l1,...,ln:</i>	<i>local variables</i>
<i>aexp:</i>	<i>returned value</i>

**Abbreviation:**

```
fun name [...]  
loc ...  
// programsegment  
ret aexp  
endf
```

**Explanation:** In addition to PROCEDURES, FUNCTIONS are the most important components of structured programming. They enable repeated evaluation of both arithmetic and string expressions and incorporation of the result of these calculations into another expression.

A FUNCTION is invoked from a program with @functionname or with FN functionname.

The body of a **FUNCTION** is composed of the declaration (**FUNCTION**), function name, parameter list, definition of local variables (**LOCAL...**), subroutine with a return value (**RETURN aexp**) and function end (**ENDFUNC**).

The statement **FUNCTION test.[parameter list]** declares a function named "test" and, optionally, a parameter list. The parameter list must start with a list of variables - separated by commas - which are to be passed to the function as 'Call by Value' variables (see Parameter list). Should the list also contain additional arrays, and/or 'Call by Reference' variables, the last 'Call by Value' variable must be followed by a comma and a **VAR**.

This is then followed by the names of the arrays with brackets and 'Call by Reference' variables.

The declaration line is followed by a program line with definitions of local variables (see Parameter list). This line begins with the **LOCAL** command.

Should a function need several local variables, they are all listed after **LOCAL** and separated by commas. Several **LOCAL** lines are allowed.

The definition of local variables is followed by the program lines that comprise the function. The result of the function calculation is returned by **RETURN aexp**.

The function must end with **ENDFUNC** command. This terminates the function declaration and returns to the program line immediately following the calling program line.

**Example:**

```
DATA 12,3,4,5,6,7,8,9,12,14
DIM a(7)
FOR i%=0 to 7
  READ a(i%)
NEXT i%
xq=@am(7,a())
sq=0
FOR i%=0 TO 7
  ADD sq,(a(i%)-xq)^2
NEXT i%
DIV sq,7
PRINT "The standard deviation is: ";sq
END
FUNCTION am(n% VAR vector())
  LOCAL i%,s
  IF n%<0
    RETURN error_code
  ELSE IF n%=0
    RETURN vector(0)
  ENDIF
  FOR i%=0 TO n%
    ADD s,vector(i%)
  NEXT i%
  RETURN DIV(s,SUCC(n%))
ENDFUNC
```

```
// This program calculates the standard deviation of
// a set given on the DATA lines. To determine this
// value the algo-rithm requires the calculation of
// an arithmetic mean. Since this formula is also
// used in other statistical calculations the arith-
// metic mean is obtained using a FUNCTION.
```

## Commands and functions

---

### Remarks:

In contrast to PROCEDURES, FUNCTIONS always return a value. This value can, if required, again be a part of an expression. The following example shows another algorithm to calculate the standard deviation, where the return value from the function (arithmetic mean) is a part of another expression.

```
DATA 12,3.4,5,6.7,8,9,12,14
DIM a(7)
sq=0
FOR i%=0 to 7
  READ a(i%)
  ..ADD sq,a(i%)^2
NEXT i%
value=DIV(sq-8*@am(7,a())),7)
PRINT "The standard deviation is: ";value
END

FUNCTION am(n% VAR vector())
  LOCAL i%,s
  IF n%<0
    RETURN error_code
  ELSE IF n%=0
    RETURN vector(0)
  ENDIF
  FOR i%=0 TO n%
    ADD s,vector(i%)
  NEXT i%
  RETURN DIV(s,SUCC(n%))
ENDFUNC
```

### See

### Also:

PROCEDURE...RETURN,  
FUNCTON...Name\$...ENDFUNC, DEFFN

## FUNCTION Name\$...ENDFUNC...Structure

**Action:** evaluates a string expression which is repeatedly used throughout the program, whereby the result of the expression changes depending on string variables passed to it.

**Syntax:**

```
FUNCTION name$ [(v1,...vn,VAR arr1(),...arrn(),
                r1,...,rn)]
    LOCAL l1,...ln
    // programsegment
    RETURN sexp
ENDFUNC
```

<i>name:</i>	<i>function name</i>
<i>v1,...,vn:</i>	<i>Call by Value variables</i>
<i>arr1(),...,arrn():</i>	<i>arrays</i>
<i>r1,...,rn:</i>	<i>Call by Reference variables</i>
<i>l1,...,ln:</i>	<i>local variables</i>
<i>sexp\$:</i>	<i>returned string expression</i>

**Abbreviation:**

```
func name$ [...]  
loc ...  
ret sexp  
endf
```

**Explanation:** This function is the same as FUNCTION...name...ENDFUNC except that the returned result is a character string.

**Example:**

```
n%=5  
FOR i%=1 TO 5  
    @build_random_string$(10,a$)  
    PRINT a$  
NEXT i%
```

## Commands and functions

---

```
END
FUNCTION build_random_string$(n%,a$)
    LOCAL i%
    a$=CHR$(RANDOM(26)+65)
    FOR i%=1 TO n%
        a$=a$+CHR$(RANDOM(26)+65)
    NEXT i%
    RETURN a$
ENDFUNC
```

```
// Prints five random character strings composed of
// 10 capital letters.
```

**Remarks:**

-

**See**

**Also:**

PROCEDURE...RETURN,  
FUNCTION...Name...ENDFUNC, DEFFN



### GET Command

**Action:** saves a portion of the screen in a string variable.

**Syntax:** GET *x1,y1,x2,y2,screensegment\$*  
*x1,y1,x2,y2:* *iexp*  
*screensegment\$:* *svar*

**Abbreviation:** -

**Explanation:** GET *x1,y1,x2,y2,screensegment\$* copies a portion of the screen with coordinates *x1,y1* (upper left corner) and *x2,y2* (lower right corner) to the string variable *screensegment\$*.

The size of such screen cut-out is limited to 32 K (maximum string length). The function GETSIZE is used to calculate the required string length.

**Example:** DEFFILL 5  
PBOX 10,10,100,100  
GET 20,20,50,50,a\$

// Draws a filled rectangle and copies a portion of  
// this rectangle to a\$.

**Remarks:** The screen segments obtained with GET can be copied back to the screen by using PUT.

**See**

**Also:** PUT



## GET # Command

**Action:** reads a record from a random access file.

**Syntax:** GET #*n*[*record*]  
*n*: *iexp*; channel number  
*record*: *iexp*

**Abbreviation:** -

**Explanation:** GET # reads a record from an R-file through the channel *n* (from 0 to 99), previously opened with OPEN. *record* is an optional parameter and contains a value between 1 and the number of records within the file. If *record* is not specified the next record in file is always read. Otherwise the record specified in *record* is read.

**Example:**

```
OPEN "R",#1,"A:\Addresses.DAT",62
FIELD #1,24 AS name$,24 AS str$,2 AT(*postcode&),12
AS city$
//
FOR i%=1 TO 5
  INPUT "NAME      : ";n$
  INPUT "Street    : ";s$
  INPUT "Postcode: ";postcode&
  INPUT "City      : ";o$
  LSET name$=n$
  LSET str$=s$
  LSET city$=o$
  PUT #1,i%
  CLS
NEXT i%
CLOSE #1
//
```

## Commands and functions

---

```
OPEN "R",#1,"A:\Adresses.DAT",62
FIELD #1,24 AS name$,24 AS str$,2 AT(*postcode&),12
AS city$
//
FOR i%=1 TO 5
  GET #1,i%
  PRINT "Record number: ";str$(i%,3)
  PRINT "NAME      : ";name$
  PRINT "Street   : ";str$
  PRINT "Postcode: ";postcode&
  PRINT "City     : ";city$
NEXT i%
CLOSE #1
```

```
// A channel for the random access file is opened
// first. Next, the record is divided with FIELD
// into: 24 bytes for the name, 24 bytes for the
// street, two bytes for the postal code and 12 bytes
// for the city, which all together totals 62 bytes.
// The FOR...NEXT loop writes five records to the
// file ADDRESSES.DAT on drive A. And finally, these
// records are read in using GET and dis-played on
// the screen again.
```

**Remarks:**

-

**See**

**Also:**

FIELD, PUT #, RECORD #

## GETEVENT Command

**Action:** monitors menu and window events

**Syntax:** GETEVENT

**Abbreviation:** getev

**Explanation:** GETEVENT monitors the occurrence of events in menu bars, pop-up menus and windows. In contrast to ON MENU, no PROCEDURE is invoked by GETEVENT. The relevant tests must be performed by the programmer. GETEVENT waits a maximum of 0.5 seconds.

**Example:**

```
DIM m$(20)
DATA Lissajous , Figure 1 , Figure 2 , Figure 3 ,
    Figure 4
DATA End , "" , Names , Robert , Piere , Gustav
DATA Emile , Hugo , !!
i%=-1
DO
    i%++
    READ m$(i%)           // read in the menu
    //                   entries
LOOP UNTIL m$(i%)="!!"    // marks the end
m$(i%)=""                // terminates a menu
//
SCREEN 16                // turns EGA mode on
COLOR 8
PBOX 0,0,639,349
MENU m$(i%)              // activates the menu bar
//
```

## Commands and functions

---

```
DO
  GETEVENT
  SWITCH MENU(1)
  CASE 0
    // nothing happened for half
    // a second
  CASE 1
    // keypress
  CASE 2,3
    // mouse click
  CASE 4 TO 19
    // something to do with
    // windows
  CASE 20
    // menu selection
  CASE 21
    // redraw
  ENDSWITCH
LOOP
```

**Remarks:**

-

**See**

**Also:**

ON MENU, PEEKEVENT, MENU()

## GETFIRST Graphic function

**Action:** reads the rectangle list.

**Syntax:** GETFIRST *i,x,y,w,h*  
*i,x,y,w,h:* *ivar; return value*

**Explanation:** When a window is moved or its size is changed it can overlap with one or more other windows. When that happens MENU(??) contains a redraw event. GETFIRST then builds the rectangle list of all overlapping rectangular areas and returns the coordinates of upper left corner of the first overlapping rectangle in the variables *x* and *y*, its width in *w* and its height in *h*. The coordinates of other overlapping rectangles can be obtained with GETNEXT.

**Example:**

```
SCREEN 16                      // EGA mode
OPENW 1,100,100,500,200,-1 // four, partially over-
                           // lapping windows
OPENW 2,150,150,300,100,-1
OPENW 3,110,230,30,30,0 // two of which are
//                        // without any border
//                        // elements
OPENW 4,80,80,80,80,0
FOR wind%=0 TO 3
  COLOR wind%*2,wind%*2+1
  q%=0
  CLIP OFFSET 0,0           // anchor point in the
  //                        // upper left corner
  GETFIRST #wind%,x%y%,w%,h% // for window #1
  WHILE w% | h%            // as long as a rectangle
  //                        // is available
    DEFFILL q%             // change fill pattern
    q%++
    CLIP x%,y%,w%,h%       // clip to rectangle
    PBOX 0,0,999,999       // and draw
```

## Commands and functions

---

```
        GETNEXT x%,y%,w%,h% // and then the next rec-
        //                 tangle
    WEND
NEXT wind%
REPEAT // wait for a mouse button
UNTIL MOUSEK
SCREEN 3
//
// A demo program to return the rectangle list for a
// window.
// The program opens four windows, and then
// determines the visible rectangle for window #1.
// By using DEFFILL, CLIP and PBOX these rectangles
// are then singled out. By changing the first para-
// meter in GETFIRST the rectangle lists for other
// windows can also be singled out.
```

**Remarks:**

-

**See**

**Also:**

RC\_INTERSECT(), GETNEXT

## GETNEXT Graphic function

**Action:** reads the rectangle list.

**Syntax:** GETNEXT x,y,w,h  
*x,y,w,h:      ivar; return value*

**Explanation:** When a window is moved or its size is changed it can overlap with one or more other windows. When that happens MENU(??) contains a redraw event. GETFIRST then builds the rectangle list of all overlapping rectangular areas and returns the coordinates of upper left corner of the first overlapping rectangle in the variables x and y, its width in w and its height in h. The coordinates of other overlapping rectangles can be obtained with GETNEXT.

**Example:** see GETFIRST

**Remarks:** -

**See**

**Also:** RC\_INTERSECT(), GETFIRST

### GETSIZE() Function

**Action:** determines the size of a rectangle in graphic mode.

**Syntax:** GETSIZE (x1,y1,x2,y2)  
*x1,y1,x2,y2: iexp*

**Explanation:** GETSIZE(x1,y1,x2,y2) serves to determine the amount of memory needed by the rectangle with the coordinates x1,y1 (upper left corner) and x2,y2 (lower right corner). GETSIZE should always be used before a GET command to assure that the relevant rectangle does not exceed the maximum string length of 32768 bytes.

**Example:**

```
SCREEN 18
a%=GETSIZE(100,100,300,200)
SCREEN 3
PRINT a%           // prints 10510.
```

**Remarks:** -

**See**

**Also:** LEN()



## GOSUB Command

**Action:** unconditional branch

**Syntax:** GOSUB procedurename[(parameterlist)]  
*procedurename:* name of a subroutine declared  
using PROCEDURE

**Abbreviation:** @

**Explanation:** The GOSUB procedurename command jumps to the named PROCEDURE without any additional condition tests. If the procedure has not been declared correctly (for example invalid name or parameter list), an error is reported.

@procedurename or even just procedurename can be used instead of GOSUB procedurename.

**Example:**

```
Do
  INPUT "Enter text ";a$
  EXIT IF a$=""
  GOSUB center_text(a$) // PROCEDURE-call
  // @center_text(a$) // PROCEDURE-call center_text(a$)
  // PROCEDURE-call
LOOP
END
PROCEDURE center_text(a$)
  LOCAL l%,x_pos%
  l%=LEN(a$)
  x_pos%=DIV(SUB(80,l%),2)
  LOCATE 5,x_pos%
  PRINT a$
RETURN

// In this example, a procedure 'center-text' is
// called from an endless loop.
```

## Commands and functions

---

**Remarks:** -

**See**

**Also:** ON...GOSUB, ON...event...GOSUB, @, FN, GOTO

## GOTO Command

**Action:** unconditional branch

**Syntax:** GOTO mar  
*mar: user defined marker*

**Abbreviation:** got mar

**Explanation:** Markers are positions within the GFA-BASIC program, used by RESTORE and GOTO. RESTORE mar is always used together with the DATA lines. GOTO mar is an unconditional jump to a previously defined marker mar.

GOTO can jump either to a mark within the main program or within a PROCEDURE or FUNCTION. GOTO's between PROCEDURES and/or FUNCTIONS are not allowed, and jumps in or out of FOR...NEXT loops are also forbidden.

**Example:**

```
SCREEN 3
PRINT "GOTO example"
PRINT
PRINT "The program is at position 1"
GOTO p2
PRINT "The program is at position 2"
p2:
PRINT "The program is at position 3"
REPEAT
UNTIL LEN(INKEY$)
```

## Commands and functions

---

**Remarks:** DO NOT USE IF POSSIBLE!!

The statistical research with students at Damogran University has shown the ratio of  $2^{276709}$  to 1 that a program using GOTOs will not run flawlessly or is not readable.

**See**

**Also:** GOSUB, EXIT IF

## GRAPHMODE Graphic command

**Action:** control of graphic output on bit level

**Syntax:** GRAPHMODE *n*  
*n*: *iexp*

**Abbreviation:** gr *n*

**Explanation:** GRAPHMODE defines the relationship between the graphic output and the screen. This relationship involves the bit-wise combination of the current screen contents and the new graphic which is to be drawn. The parameter *n* specifies how this combination is to be performed. Four modes are possible:

<i>n</i>	Rule	Effect
1	set	The new bit pattern is moved to the current screen.
2	OR	All points which are set in, either the current screen or the new pattern, are set.
3	XOR	Only the points which are different in both the current screen and the new pattern are set.
4	AND	All points which are set in both the current screen and the new pattern are set.

GRAPHMODE 1 is default

**Example:**

```
SCREEN 16           // EGA mode
BOX 10,10,100,200   // Graphmode 1 is default
BOX 15,15,105,205
INTR(33,_AH=8)      // wait for a keypress
```

## Commands and functions 8-470

---

```
CLS
GRAPHMODE 2           // transparent
BOX 10,10,100,200
BOX 15,15,105,205
~INTR(33,_AH=8)       // wait for a keypress
CLS
GRAPHMODE 3           // inverse
BOX 10,10,100,200
BOX 15,15,105,205
~INTR(33,_AH=8)       // wait for a keypress
CLS
GRAPHMODE 4           // inverse and
                        // transparent
BOX 10,10,100,200
BOX 15,15,105,205
~INTR(33,_AH=8)       // wait for a keypress
EDIT                  // draws two overlapping rec-
                        // tangles
```

**Remarks:** -

**See**

**Also:** -



### HARDCOPY Command

**Action:** screen dump

**Syntax:** HARDCOPY

**Abbreviation:** har

**Explanation:** HARDCOPY calls interrupt \$5 which is also invoked when the Print key is pressed.

**Example:**

```
SCREEN 3
FOR i%=1 TO 300
  PRINT i%
NEXT i%
HARDCOPY
```

**Remarks:** -

**See**  
**Also:** -



## HEX\$() Function

**Action:** converts an integer expression to hexadecimal representation.

**Syntax:** HEX\$(m[,n])  
*m,n: iexp*

**Explanation:** After conversion the hexadecimal representation of integer expression *m* is returned as a plain string.

The parameter *n* is optional and determines how many places should be used to represent the number. If *n* is greater than the number of places needed to represent *m* the converted number is padded with leading zeros.

**Example:**

```
PRINT HEX$(25)           // prints 19
PRINT HEX$(1001,6)       // prints 0003E9
```

**Remarks:** -

**See**

**Also:** BIN\$(), OCT\$(), DEC\$()

### HTAB Command

**Action:** cursor positioning

**Syntax:** HTAB *column*  
*column:* *iexp*

**Abbreviation:** -

**Explanation:** Places the cursor in column specified in column.

**Example:** PRINT AT(1,1);"Hello GFA"  
HTAB 20  
PRINT "Hello GFA"

```
// Prints Hello GFA from the first column on the  
// first line, and then prints the same string again  
// only from the 20th column.
```

**Remarks:** -

**See**

**Also:** LOCATE, PRINT AT, VTAB, TAB



### IF...ENDIF Conditional directive

**Action:** a conditional branch statement allowing for execution of specific program segments only when a condition is logically "true".

**Syntax:**

```
IF condition [THEN]
    // programsegment
[ELSE IF condition
    // programsegment]
[ELSE
    // programsegment]
ENDIF
```

*Condition: any numeric, logic or string condition*

**Abbreviation:**

```
if
    // programsegment
[el if]
[e]
endi
```

**Explanation:** The IF...ENDIF directive is, in addition to SELECT...CASE, the most important command for controlling the program flow. The program segment after an IF...ENDIF statement will be executed if, and only if, the condition immediately following the IF is logically "true". Otherwise, the control is passed to an ELSE...IF or ELSE within the same IF...ENDIF structure. If there are no ELSE...IF or ELSE, a branch is performed to the statement immediately after the next ENDIF.

The structure

```
IF condition x
    // programsegment A
ELSE
    // programsegment B
ENDIF
```

is an exclusive structure. This means that the test, if the condition x is logically "true", will be performed first. If it is, the program segment A is executed and a branch to the statement following the ENDIF is taken. If the condition x is logically "false", the program segment B is executed and a branch to the statement following the ENDIF is taken. In no case will both program segments be executed!

The structure

```
IF condition x
    // program segment A
ELSE IF condition y
    // programsegment B
ELSE IF Condition z
    // programsegment C
ELSE
    // programsegment D
ENDIF
```

allows for an array of exclusive tests.

The first condition in the condition list x, y or z which is logically "true" causes the execution of the corresponding program segment (A, B or C), and then a branch to the statement immediately after ENDIF. This is irrespective of whether only one, several or all conditions in the condition list x, y and z is/are logically "true". Only the program segment after the first

"true" condition is executed, followed by an immediate jump to the statement after ENDIF!

If none of the x, y or z conditions are logically "true", the program segment D after ELSE is executed and the IF...ENDIF structure is completed.

### Example:

```
a%=10
IF a%
    PRINT "a%<>0"
ENDIF // gives a%<>0

IF MOD(42,6)
    PRINT "42 is not fully divisible by 6"
ELSE
    PRINT "42 is fully divisible by 6"
ENDIF

// Gives 42 is fully dividible by 6

IF MOD(42,4)
    PRINT "42 is not fully divisible by 4"
ELSE IF MOD(42,5)
    PRINT "42 is not fully divisible by 5"
ELSE IF MOD(42,8)
    PRINT "42 is not fully divisible by 8"
ELSE IF MOD(42,9)
    PRINT "42 is not fully divisible by 9"
ELSE
    PRINT "42 is fully divisible by 4,5,8 and 9"
ENDIF

// Gives only 42 is not fully divisible by 4, because
// the very first condition is logically "true".

IF MOD(42,2)
    PRINT "42 is not fully divisible by 2"
ELSE IF MOD(42,3)
```

```
PRINT "42 is not fully divisible by 3"
ELSE IF MOD(42,6)
PRINT "42 is not fully divisible by 6"
ELSE IF MOD(42,7)
PRINT "42 is not fully divisible by 7"
ELSE
PRINT "42 is fully divisible by 2,3,6 and 7"
ENDIF
```

```
// Gives 42 is fully divisible by 2,3,6 and 7, since
// none of the preceeding ELSE conditions is logi-
// cally "true".
```

**Remarks:**

Any numeric, logic or string condition can be tested using the IF...ENDIF command. The related SELECT...CASE command is only suitable for testing of numeric conditions.

**See****Also:**

SELECT...ENDSELECT, ON...GOSUB

## IMP() Function

**Action:** a logical bit-wise combination of two bit patterns

**Syntax:** IMP(*i,j*)  
*i,j: iexp*

**Explanation:** IMP(*i,j*) combines the expressions *i* and *j* based on their order. The result is equivalent to a logical sequence. This means that something is false only when a true statement is followed by a false one. For expressions *i* and *j* this applies to their binary representation, i.e. the resulting bit will be 0 only when the corresponding bit in the first argument (*i*) is 1 and in the second argument (*j*) is 0.

**Example:**

```
PRINT BIN$(3,4)           // prints 0011
PRINT BIN$(10,4)          // prints 1010
PRINT BIN$(IMP(3,10),4)   // prints 1110
```

1110 is binary representation of number 14. However, IMP(3,10) returns -2. To understand this all 32 bits must be examined:

```
BIN$(3,32)      = 0000000000000000000000000000000000000011
BIN$(10,32)     = 00000000000000000000000000000000000001010
BIN$(IMP(3,10),32)= 1111111111111111111111111111111111111110
```

The result of IMP(3,10) is therefore -2.

**Remarks:** IMP is the only bit-wise operator for which the order of the arguments is important. This is because the result will produce a 0 only when, at the same position, a "true" (1) in the first argument is followed by a "false" (0) in the second argument.



```
BIN$(10,32)      = 00000000000000000000000000001010  
BIN$(3,32)       = 000000000000000000000000000000011  
BIN$(IMP(10,3),32)= 11111111111111111111111111110111  
IMP(10,3)        = -9
```

**Also:** AND(), OR(), XOR(), EQV()

### INC Command

**Action:** increments a numeric variable.

**Syntax:** INC x  
x: *avar*

**Abbreviation:** -

**Explanation:** INC x increments the value of x by 1.

**Example:** x=2.7  
INC x  
PRINT x // prints 3.7

**Remarks:** Although INC can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of INC

```
x=x+1
x:=x+1
x+=1
x++
SUB x,-1 or
ADD x,1
```

can be used also.

When integer variables are used INC doesn't test for overflow!

**See**

**Also:** DEC, ADD, SUB, MUL, DIV, ++, --, +=, -=, \*=, /=

## INFOW# Graphic command

**Action:** writes a string on the info line of a window.

**Syntax:** INFOW #n,a\$  
*n:* *iexp*  
*a\$:* *sexp*

**Abbreviation:** -

**Explanation:** INFOW #n,a\$ writes the string a\$ on the info line of the window n, previously opened with OPENW #n. n can have the values of 1, 2, 3 or 4.

**Example:**

```
SCREEN 16                // EGA mode
OPENW #1,10,10,250,100,-1
TITELW #1," GFA-BASIC window "
INFOW #1,"Window No. 1"
REPEAT
    a$=inkey$
UNTIL a$=chr$(27)
CLOSEW #1
SCREEN 3
EDIT

// Draws a window on the screen and writes "Window
// No. 1" on the info line of the window.
```

**Remarks:** -

**See**

**Also:** OPENW, CLOSEW, TITELW, SIZEW, TOPW, FULLW, CLEARW

### INKEY\$ Function

**Action:** reads a character from the keyboard (excluding special keys like Shift, Alt, Alt Gr, Ctrl...).

**Syntax:** INKEY\$

**Abbreviation:** -

**Explanation:** INKEY\$ does not wait for a keypress but, if no keys were pressed since the last keyboard request (by the processor), it returns a space. Otherwise, INKEY\$ reports the ASCII code of the pressed key.

If the pressed key has no ASCII code (the special keys, for example), the scan code of the pressed key is returned instead. If this is this case a two character string is returned, the first of which is a CHR\$(0) and the second the corresponding key code.

**Example:**

```
D0
  t$=INKEY$
  IF t$<>" "
    IF LEN(t$)=1      // normal key
      PRINT "Key: ";t$;SPC(3);"ASCII code : ";ASC(t$)
    ELSE
      PRINT "CHR$(0), Scan code : ";CVI(t$)
    ENDIF
  ENDIF
LOOP

// This example displays the ASCII or scan code for
// each pressed key.
```

**Remarks:** -

**See**

**Also:** INPUT(), INPUT #, INP , LINE INPUT, FORM  
INPUT

### INP(#n) Command

**Action:** reads a byte from a previously opened file.

**Syntax:** INP(#n)  
*n: iexp; channel number*

**Abbreviation:** -

**Explanation:** INP(#n) reads a byte from a previously opened file. The numerical expression *n* contains the channel number (from 0 to 99), with which the file is being accessed.

**Example:**

```
OPEN "i",#1,"A:\TEST.DAT"
FOR i%=1 to 20
  a&=INP(#1)
  PRINT a&
NEXT i%
CLOSE #1
```

```
// Opens the file TEST.DAT on drive A, reads one byte
// from this file inside a FOR...NEXT loop 20 times
// and displays this value on the screen.
```

**Remarks:** -

**See**  
**Also:** OUT

## INP(PORT) Function

**Action:** reads a byte from a port.

**Syntax:** INP(PORT n)  
*n: iexp; port number*

**Explanation:** INP(PORT n) reads a byte from a hardware port register, RTC for example.

**Example:** This command implies an intimate knowledge of the hardware and is not portable.

**Remarks:** INP(^ n), INP|(PORT n) and INP|(^n) are synonymous with INP(PORT n) and can be used instead.

Together with 16 bit IN command, INP&(^n) can be used to read a word (two bytes) and INP%(PORT n) a longword (four bytes) from successive port addresses.

**See**

**Also:** OUT PORT

### INPUT Command

**Action:** entry of variables or lists of variables, with or without the prompt

**Syntax:** INPUT ["Text",]x[,y,...] or  
INPUT ["Text";]x[,y,...]  
*Text: any text as prompt*  
*x,y: avar or svar*

**Abbreviation:** -

**Explanation:** INPUT always starts from the last cursor position. To define the location where the input should take place, the cursor can be positioned using PRINT AT followed by a semi-colon, LOCATE, VTAB or HTAB.

If INPUT contains the optional prompt, this prompt is separated from the following variables by a comma or a semi-colon. If a semi-colon is used, a question mark and a space are appended to the text of the prompt and the cursor is placed behind them. If a comma is used the cursor is placed immediately after the last character of the prompt text. If INPUT does not contain a prompt, a question mark and a space appear followed by the cursor.

If only one variable is requested, its input must be ended by pressing the Return or the Enter key. If INPUT contains a list of variables the entry of each individual variable is terminated by pressing the Return or the Enter key. It is also possible to separate individual variables in the list with commas and confirm them all with one single press of the Return or the Enter key. The corrections within the variable list are made by using the Backspace, Delete and Insert keys, as well as the cursor keys.



The string variables following INPUT cannot contain commas. If they do LINE INPUT should be used instead. The maximum input length is 255 characters.

**Example:**

```
INPUT a$  
INPUT "",b$  
INPUT "Two digits, please ";x,y  
PRINT a$'b$'x'y
```

```
// Two strings and two numeric variables are read in  
// the above example. The first INPUT command is  
// prompted with a '?', the second without any text  
// and the third with a request 'Two digits, please:  
// ? '.
```

**Remarks:**

In case of incorrect input (for example a string was entered when a numeric variable was expected), an alarm signal is sounded and the input must be done from the beginning again.

The special characters can be entered in three different ways:

by holding down the Alternate key and typing from the numeric keypad,

by holding down the Control and the s keys and entering an additional character,

by holding down the Control and the a key and entering the ASCII code of the character.

**See**

**Also:**

LINE INPUT, FORM INPUT, INPUT #, LINE INPUT #, FORM INPUT #

## INPUT\$ Command

**Action:** reads a string from the keyboard or a file.

**Syntax:** INPUT\$(count[,#n])  
*count:* *exp; the number of characters to read*  
*n:* *exp; channel number*

**Abbreviation:** -

**Explanation:** INPUT\$ reads a string from the keyboard composed of count characters. By supplying the optional channel number n (from 0 to 99) the characters can also be read from a previously opened file. In either case, count specifies the number of characters to read. The input is not echoed to the screen so this command can be used for password entry, for example.

**Example:**

```
OPEN "o",#1,"A:\TEST.DAT"
PRINT #1, "TEST...TEST...TEST"
CLOSE #1
//
OPEN "i",#1,"A:\TEST.DAT"
a$=INPUT$(6,#1)
CLOSE #1
PRINT v$ // prints TEST..

PRINT "Please enter your name"
PRINT INPUT$(7)

// Reads seven characters from the keyboard and dis-
// plays them on the screen.
```

**Remarks:** -

**See**

**Also:** INPUT #, LINE INPUT #

## INPUT # Command

**Action:** reads data from a previously opened file.

**Syntax:** INPUT #n,v1[,v2,...]  
*n:* *exp; channel number*  
*v1,v2,...:* *avar or svar*

**Abbreviation:** -

**Explanation:** INPUT #n reads data from a file, accessed with the channel number n (from 0 to 99). Either individual values or whole variable lists can be read, the latter using commas to separate the individual variables in the list.

**Example:**

```
OPEN "i",#1,"A:\TEST.TXT"
INPUT #1,a$,b$
CLOSE #1
PRINT a$,b$

// Reads two strings from file TEST.TXT on drive A
// and writes them to the screen.
```

**Remarks:** INPUT # is equivalent to INPUT, except that INPUT # isn't normally used to read from the keyboard.

**See**

**Also:** INPUT\$, LINE INPUT #, INPUT, LINE INPUT

### INSERT Command

**Action:** inserts a numeric or a string expression at the specified place in a one-dimensional array of corresponding variable type.

**Syntax:** `INSERT x(m)=y`  
*m:* `iexp`  
*y:* `aexp`, if *x()* is a numeric array or  
`sexp`, if *x()* is a string array  
*x():* a one-dimensional array of any variable type

**Abbreviation:** `ins a(m)=y`

**Explanation:** `INSERT x(m)=y` inserts *y* in array *x()* at position *m*. In other words all items in array *x()* whose indices are greater than or equal to *m* are moved one position down. The last element in *x()* is deleted with each `INSERT`.

**Example:**

```
DIM a$(4)
a$(1)="GFA BASIC MS DOS"
a$(2)="GFA BASIC OS/2"
a$(3)="MS PDS 7.0"
a$(4)="TURBO BASIC"
INSERT a$(3)="GFA BASIX"
FOR i%=1 to 4
    PRINT a$(i%)
NEXT i%
```

```
// prints
// GFA BASIC MS DOS
// GFA BASIC OS/2
// GFA BASIX
// MS PDS 7.0
```

**Remarks:** -

**See**

**Also:** DELETE

### INSTR() Function

**Action:** searches a string expression for occurrence of a substring, optionally from a given position. If the substring is found, the position at which it begins is returned. If the substring is not found a 0 is returned.

**Syntax:** INSTR(a\$,b\$[,m%]) or  
INSTR([m%],a\$,b\$)  
*a\$,b\$:* *sexp*  
*m%:* *iexp*

**Abbreviation:** -

**Explanation:** INSTR(a\$,b\$,m%) searches through the string expression a\$ starting from position m% for the substring b\$. If m% is not given, the search starts from the first character in string a\$. If b\$="" the command returns 0.

**Example:**

```
PRINT INSTR("Hello GFA","ll",2)    // prints 3
PRINT INSTR("Hello GFA","ll")      // prints 3
PRINT INSTR("Hello GFA","ll",4)    // prints 0
```

**Remarks:** -

**See**

**Also:** RINSTR()

## INT() Function

**Action:** returns the integer portion of a numeric expression.

**Syntax:** INT(*x*)  
*x*: *aexp*

**Explanation:** INT(*x*) returns the first smaller integer, i.e. it is equivalent to rounding down.

**Example:**

```
PRINT INT(3*1.2)      // prints    3
PRINT INT(-3*1.2)     // prints   -4
PRINT FLOOR(3*1.2)    // prints    3
PRINT FLOOR(-3*1.2)   // prints   -4
```

**Remarks:** FLOOR() is synonymous with INT() and can be used instead.

**See**

**Also:** FLOOR(), CEIL(), TRUNC(), FIX(), FRAC()

### INT{} Function

**Action:** reads a word (16 bits) from an address.

**Syntax:** INT{addr}  
*addr: address*

**Explanation:** Reads a word (16 bits) from an address.

**Example:**

```
PRINT INT{$b800:0}    // prints a character with
// attribute from text
// screen memory.
//
PRINT WORD{$b800:0}   // prints a character with
// attribute from text
// screen memory.
```

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

WORD{} and SHORT{} are synonymous with INT{} and can be used instead.

addr: see the {} function.

**See**

**Also:** DPEEK(), BYTE{}, CARD{}, WORD{}, LONG{}, {}, SINGLE{}, DOUBLE{}, SHORT{}, USHORT{}, UWORD{}



## INT{} = Command

**Action:** writes a word (16 bits) to an address.

**Syntax:** INT{addr} = m  
*addr: address*  
*m: iexp*

**Abbreviation:** -

**Explanation:** Writes a word (16 bits) to an address.

**Example:** -

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

WORD{} = and SHORT{} = are synonymous with INT{} = and can be used instead.

addr: see the {} function.

**See**

**Also:** DPOKE(), BYTE{} =, CARD{} =, WORD{} =, LONG{} =, {} =, SINGLE{} =, DOUBLE{} =, SHORT{} =, USHORT{} =, UWORD{} =

### INTR() Function

**Action:** invokes a DOS or a BIOS function.

**Syntax:**        ~ INTR(i[j,k])  
                  *i:    number of the interrupt*  
                  *j,k:  functions and sub-functions*

**Explanation:** In principle, all DOS and BIOS functions can be invoked using ~ INTR(i[j,k]). The parameter i specifies the number of the interrupt.

Interrupts get their parameters from registers (\_AX...). These can be set before the call (\_AH=0) or directly within the INTR function following the interrupt number (INTR(\$16,\_AH=0)).

Available registers are:

\_AX, \_BX, \_CX, \_DX, \_SP, \_BP, \_SI, \_DI, \_IP, \_FL,  
\_CS, \_DS, \_SS or \_ES.

The result is returned in:

\_CS, \_IP, \_SP, \_SS and \_FL.

**Example:**

```
DO
  _AH=0           // The high byte of the AX
  //             register is loaded with 0.
  ~INTR($16)      // The BIOS function 22
  //             ($16) is invoked.
  IF _AL=0        // The low low byte of the
  //             AX register is examined
    PRINT "SCAN-CODE : ";_AH
  ELSE
    PRINT "ASCII-CODE : ";_AL
  ENDIF
LOOP UNTIL _AL=27
```

The above example calls the BIOS interrupt function \$16 (keyboard input). The function (`_AH=0`) reads a character from the keyboard. After the interrupt, the function returns the ASCII code of the pressed key in the high byte of AX and the scan code in the low byte of AX. In the above example, the high byte of AX is initialized to 0 with the assignment `_AH=0` after the DO. Following that the function is invoked. If a special key is pressed (a key without ASCII code), the low byte of AX contains the value of 0. This is read with `_AL=0` in the IF statement. If a special key was pressed the IF condition reads the scan code of the key from the high byte of AX by using `_AH`.

Otherwise the ELSE branch reads the ASCII code of the key from the low byte of AX by using `_AL`. The loop is terminated by pressing the Esc key.

This example can be modified so that the expected value from the high byte in AH is written directly into the parameter list of the interrupt call:

```
DO
  ~INTR($16,_AH=0)
  IF _AL=0
    PRINT "SCAN-CODE : ";_AH
  ELSE
    PRINT "ASCII-CODE : ";_AL
  ENDIF
LOOP UNTIL _AL=27
```

The following example tests the status in the keyboard flag byte.

```
DO
  ~INTR($16,_AH=2)
  IF BTST(_AL,0)
    PRINT "right shift key was pressed"
  ENDIF
```

## Commands and functions

---

```
IF BTST(_AL,1)
    PRINT "left shift key was pressed"
ENDIF
IF BTST(_AL,2)
    PRINT "Ctrl key was pressed"
ENDIF
IF BTST(_AL,3)
    PRINT "Alt key was pressed"
ENDIF
IF BTST(_AL,4)
    PRINT "Scroll-Lock key is on"
ENDIF
IF BTST(_AL,5)
    PRINT "Num-Lock key is on"
ENDIF
IF BTST(_AL,6)
    PRINT "Caps-Lock key is on"
ENDIF
IF BTST(_AL,7)
    PRINT "Insert mode is on"
ENDIF
LOOP UNTIL BTST(_AL,5)=0
```

**Remarks:** -

**See**  
**Also:** -



# KEYGET Function

**Action:** gets the first character in the keyboard buffer.

**Syntax:** KEYGET *n*  
*n*: *ivar*

**Abbreviation:** keyg *n*

**Explanation:** KEYGET *n*% waits for a keypress and returns in *n*% a longword with the following layout:

Bit 0 to 7	ASCII code
Bit 8 to 15	scan code

**Example:** KEYGET *n*%  
PRINT HEX\$(*n*%,4)

```
// Waits for key entry and then returns the codes of  
// the pressed key.
```

**Remarks:** -

**See**

**Also:** INKEY\$, KEYTEST, KEYLOOK

## KEYTEST Function

**Action:** reads the first character in the keyboard buffer.

**Syntax:** KEYTEST n  
*n: ivar*

**Abbreviation:** keyt n

**Explanation:** KEYTEST n is similar to INKEY\$, that is to say, it reads a character from the keyboard when a key - other than Alt, Ctrl, Shift or Caps Lock - is pressed. If no key was pressed a 0 is returned. Otherwise the ASCII code of the character is returned (for layout see KEYGET).

**Example:**

```
REPEAT
  KEYTEST n%
UNTIL BYTE(n%)=27

// The program loops until Esc is pressed.
```

**Remarks:** If INKEY\$ were used the same loop would look like this:

```
REPEAT
UNTIL INKEY$=CHR$(27)

or

REPEAT
UNTIL ASC(INKEY$)=27
```

**See**

**Also:** INKEY\$, KEYGET, KEYLOOK

# KILL Command

**Action:** deletes a file

**Syntax:** KILL path\$  
*path\$:* *sexp; path name*

**Abbreviation:** -

**Explanation:** KILL path\$ deletes the file whose pathname is given in path\$.

**Example:**

```
path$="A:\TEST.TXT
IF EXIST path$
  KILL path$
ENDIF

// Checks if the file with the name TEST.TXT exists
// on drive A and deletes it.
```

**Remarks:** KILL path\$ always deletes one file only, even when path\$ ends with "\*\*.\*".

**See**

**Also:** -



## KILLEVENT Command

**Action:** deletes redraw messages.

**Syntax:** KILLEVENT

**Abbreviation:** -

**Explanation:** KILLEVENT clears the redraw message buffer.

**Example:** MOVEW #1,100,100  
KILLEVENT

```
// This program segment moves a window and draws the
// background under the old window position again.
// The redraw messages can be ignored in certain
// circumstances, the movement of several windows for
// example, or due to other reasons which cause the
// new drawing of the windows.
```

**Remarks:** -

**See**

**Also:** ON MENU, GETEVENT, PEEKEVENT, MENU()

### LCASE\$() String function

**Action:** converts all capital letters of a string expression to lower case letters, with the exception of PC umlauts (US character table).

**Syntax:** LCASE\$(a\$)  
*a\$: sexp*

**Abbreviation:** -

**Explanation:** -

**Example:** PRINT LCASE\$("HELLO GFA, ÖÖÖHA")

```
// prints  
// hello gfa, öööha
```

**Remarks:** -

**See**

**Also:** UPPER\$(), LOWER\$(), UCASE\$(), XLATE\$()

## LEFT\$() Function

**Action:** returns the first characters of a string expression.

**Syntax:** LEFT\$(a\$,m%)  
*a\$:* *sexp*  
*m%:* *iexp*

**Abbreviation:** -

**Explanation:** LEFT\$(a\$,m%) returns the first m% characters of the string expression a\$. If m% is greater than the number of characters in a\$ (spaces and CHR\$(0) are characters too!), the whole a\$ is returned. If m% is less than 1 or if m% is not given, the first character of a\$ is returned.

**Example:**

```
PRINT LEFT$("Hello GFA",5) // prints Hello
PRINT LEFT$("Hello GFA",20) // prints Hello GFA
PRINT LEFT$("Hello GFA",-1) // prints H
PRINT LEFT$("Hello GFA") // prints H
```

**Remarks:** -

**See**

**Also:** RIGHT\$(), MID\$()

### LEN() String function

**Action:** determines the length of a character string.

**Syntax:** LEN(a\$)  
*a\$: sexp*

**Explanation:** Determines the number of characters contained within a string expression and returns this value.

For GFA-BASIC TYPEs and TYPE variables, LEN(type:) and LEN(var.) return the length of the TYPEs and the TYPE variables respectively.

**Example:**

```
PRINT LEN("Hello world") // prints 11
PRINT LEN(" Hello world ")// prints 13
TYPE test:
-CHAR*20 t$
ENDTYPE
test:t1.
PRINT LEN(text:)'LEN(t1.)// prints 20 20
```

**Remarks:** -

**See**

**Also:** -

## LET Command

**Action:** assignment of variables

**Syntax:** LET  $x = y$   
*x:* avar or svar  
*y:* aexp or sexp

**Abbreviation:** -

**Explanation:** LET  $x = y$  command assigns the variable  $x$  with the value in expression  $y$ .  $x$  and  $y$  must either be both numeric or both strings.

LET  $x = y$  is normally not necessary, but is used when one of the reserved GFA-BASIC variables (for example DATA) needs to be assigned a value.

**Example:** LET DATA\$=STR\$(PI,9)

**Remarks:** -

**See**

**Also:** -

### LINE Graphic command

**Action:** draws a line on the screen.

**Syntax:** LINE x1,y1,x2,y2  
x1,y1,x2,y2: *iexp*

**Abbreviation:** li x1,y1,x2,y2

**Explanation:** LINE x1,y1,x2,y2 draws a line on the screen from the point with coordinates x1,y1 to the point with coordinates x2,y2. The origins of the coordinate system are in the upper left corner of the screen.

**Example:**

```
CLS
FOR i%=0 TO 100 STEP 2
    LINE 10,10,ADD(100,i%),ADD(100,i%)
NEXT i%
```

// Draws lines as rays emanating from 10,10.

**Remarks:** The style and colour of the line can be defined using **DEFLINE** and **COLOUR** commands.

**See**

**Also:** DRAW TO

## LINE INPUT Command

**Action:** entry of string variables or lists of string variables with or without a prompt.

**Syntax:** LINE INPUT ["Text"],a\$,b\$,... or  
LINE INPUT ["Text";]a\$,b\$,...  
*Text:* any text  
*a\$,b\$:* svar

**Abbreviation:** -

**Explanation:** The same as INPUT, except that LINE INPUT works only with strings. Another feature of LINE INPUT is that the strings can contain commas.

**Example:** LINE INPUT a\$  
INPUT b\$  
PRINT a\$'b\$

// Enter the string "Kom,ma" twice. The PRINT command  
// will display Kom,ma Kom.

**Remarks:** see INPUT.

**See**

**Also:** INPUT, FORM INPUT, INPUT #, LINE INPUT #,  
FORM INPUT #

### LINE INPUT # Command

**Action:** reads strings from a previously opened file.

**Syntax:** LINE INPUT #*n*,*v1*\$[,*v2*\$,...]  
*n*: *xp*; *channel number*  
*v1*\$,*v2*\$,...: *svar*

**Abbreviation:** l input #....

**Explanation:** LINE INPUT #*n* reads strings from a file, accessed with the channel number *n* (from 0 to 99). Either individual strings or whole lists of strings can be read, the latter using commas to separate the strings in the list.

**Example:**

```
OPEN "i",#1,"A:\TEST.TXT"  
LINE INPUT #1,a$,b$  
CLOSE #1  
PRINT a$,b$
```

```
// Reads two strings from the file TEST.TXT on drive  
// A and writes them to the screen.
```

**Remarks:** LINE INPUT # is equivalent to INPUT # except that LINE INPUT # can only read strings, while INPUT # can read numeric variables as well.

**See**

**Also:** INPUT\$, INPUT #, INPUT, LINE INPUT



## LIST Command

**Action:** displays the current program.

**Syntax:** LIST [p\$]  
*p\$:* *sexp*

**Abbreviation:** li p\$

**Explanation:** LIST prints the program currently in memory to screen or to a file.

**Example:** LIST "C:\BASIC.GFA\TEST.LST"

```
// Saves the current program in the subdirectory  
// BASIC.GFA on partition C under the name  
// TEST.LST.
```

**Remarks:** -

**See**

**Also:** -

### LOAD Command

**Action:** loads a GFA-BASIC program into GFA-BASIC editor.

**Syntax:** LOAD p\$  
*p\$*: *sexp*

**Abbreviation:** loa p\$

**Explanation:** LOAD p\$ can only be used from direct mode or by selecting the corresponding GFA-BASIC editor function and it will load the GFA-BASIC program specified in p\$ in main memory.

**Example:** LOAD "C:\BASIC.GFA\TEST.GFA"

```
// Loads the TEST.GFA program from subdirectory  
// BASIC.GFA in partition C.
```

**Remarks:** -

**See**

**Also:** SAVE, PSAVE

## LOADFONT Command

**Action:** loads an external font.

**Syntax:** LOADFONT p\$  
*p\$: sexp*

**Abbreviation:** loadf

**Explanation:** LOADFONT p\$ loads the file specified in p\$ which contains the bitmap of a character set and replaces the current character set with the loaded one.

**Example:** LOADFONT"C:\FONTS\TESTFONT.FNT"

**Remarks:** The FREEFONT command is used to delete the loaded fonts.

**See**

**Also:** FREEFONT

### LOC() Function

**Action:** returns the current position of the file data pointer.

**Syntax:** LOC(*#n*)  
*n*: *ixp*

**Explanation:** LOC(*#n*) works only on files previously opened with OPEN using channel *n* and returns the current position of the data pointer (locate).

**Example:**

```
OPEN "i",#1, "TEST.TXT"
DO UNTIL EOF(#1)
  INPUT #1,a$
  PRINT " ";a$,LOC(#1)
LOOP
CLOSE #1
```

```
// Opens the file TEST.TXT in current directory and
// reads its contents as well as the position of the
// data pointer until end of file.
```

**Remarks:** -

**See**

**Also:** EOF(), LOF()

## LOCAL Command

**Action:** defines local variables in a subroutine.

**Syntax:** LOCAL a,b,...  
*a,b,...: variables of any type*

**Abbreviation:** loc a,b,c...

**Explanation:**

**Example:**

```
SCREEN 3
a%=0
FOR i%=1 TO 10
  a%+=i%
  @test (a%)
NEXT i%
PRINT a%
//
PROCEDURE test(var a_r%)
  LOCAL i%
  FOR i%=1 TO 5
    a%+=i%
  NEXT i%
  RETURN

// Prints 205. The FOR...NEXT loop counter i%
// is defined both as a global and
// a local variable.
```

**Remarks:** -

**See**

**Also:** -

# LOCATE Command

**Action:** cursor positioning

**Syntax:** LOCATE row,column  
*row,column: ivar*

**Abbreviation:** loc row,column

**Explanation:** -

**Example:** LOCATE 12,4  
  
// Places the cursor on the fourth column of the  
// twelfth row.

**Remarks:** Do note that LOCATE, in contrast to PRINT AT(), specifies row first and column second.

**See**

**Also:** PRINT AT, VTAB, HTAB

## LOF() Function

**Action:** determines the length of a file.

**Syntax:**       LOF(#n)  
                  *n:    iexp*

**Explanation:**   LOF(#n) works only on a file previously opened with OPEN though the channel n and returns its length in bytes (length of file).

**Example:**

```
OPEN "i",#1, "TEST.TXT"
PRINT "File length in bytes: ";LOF(#1)
CLOSE #1

// Opens file TEST.TXT incurrent directory and
// returns its size.
```

**Remarks:**       -

**See**

**Also:**           LOC(), EOF()

### LOG() Numeric function

**Action:** returns the natural logarithm of a numeric expression.

**Syntax:** LOG(*x*)  
*x*: *aexp*

**Explanation:** LOG(*x*) calculates the logarithm of *x* to the base of EULER's number *e* (= 2.178....)

**Example:** PRINT LOG(SQR(2)) // prints 0.34657...

**Remarks:** LOG(*x*) is the reverse function of EXP(*x*), which means:

$$\text{LOG}(\text{EXP}(\text{PI})) = \text{PI} = 3.14....$$

The following equation is used to calculate the logarithm of any base:

$$\text{LOGBASIS}(x) = \text{LOG}(x) / \text{LOG}(\text{BASIS})$$

**See**

**Also:** EXP()



## LOG2() Numeric function

**Action:** returns the binary logarithm of a numeric expression.

**Syntax:** LOG2(x)  
*x:* *aexp*

**Explanation:** LOG2(x) calculates the logarithm of x to the base of 2.

**Example:** PRINT LOG2(42) // prints 5.392...

**Remarks:** LOG2(x) is the reverse function of  $2^x$ , which means:

$$\text{LOG2}(2^{\text{SQR}(2)}) = 1.414\dots$$

The following equation is used to calculate the logarithm of any base:

$$\text{LOGBASIS}(x) = \text{LOG}(x) / \text{LOG}(\text{BASIS})$$

**See**

**Also:** Exponential operator (^)

### LOG10() Numeric function

**Action:** returns the base 10 logarithm of a numeric expression.

**Syntax:** LOG10(*x*)  
*x*: *aexp*

**Explanation:** LOG10(*x*) calculates the logarithm of *x* to the base of 10.

**Example:** PRINT LOG10(SQR(2)) // prints 0.15051...

**Remarks:** LOG10(*x*) is the reverse function of  $10^x$ , which means:

$$\text{LOG10}(10^{\text{SQR}(2)}) = 1.414...$$

The following equation is used to calculate the logarithm of any base:

$$\text{LOGBASIS}(x) = \text{LOG}(x) / \text{LOG}(\text{BASIS})$$

**See**

**Also:** Exponential operator (^)

## LONG{} Function

**Action:** reads a double word (32 bits) from an address.

**Syntax:** LONG{addr}  
*addr: address*

**Explanation:** Reads a double word (32 bits) from an address.

**Example:**

```
a=1.2345
PRINT HEX$(LONG{*a},8)''HEX$(LONG{*a+4},8)
PRINT
FOR i%=0 TO 7
  PRINT HEX$(PEEK(*a+i%),2)
NEXT i%
PRINT
PRINT a
```

```
// Prints first 126E978D EFF3C083, which is the
// internal representation of a as a longword and
// then 8D 97 6E 12 83 C0 F3 3F, which is the inter-
// nal representation of a read in as bytes.
```

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

{ } or LPEEK() are synonymous with LONG{ } and can be used instead.

addr: see the { } function.

**See**

**Also:** LPEEK(), BYTE{ }, WORD{ }, CARD{ }, INT{ }, { }, SINGLE{ }, DOUBLE{ }, SHORT{ }, USHORT{ }, UWORD{ }

## LONG{} = Command

**Action:** writes a double word (32 bits) to an address.

**Syntax:** LONG{addr} = m  
*addr: address*  
*m: iexp*

**Abbreviation:** -

**Explanation:** Writes a double word (32 bits) to an address.

**Example:** LONG{\$40:\$1E}=2002 // writes 2002 at offset  
// \$1E in segment \$40

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

{ } or LPOKE() are synonymous with LONG{ } and can be used instead.

addr: see the { } function.

**See**

**Also:** LPOKE, BYTE{ }, CARD{ }, WORD{ }, INT{ }, { }, SINGLE{ }, DOUBLE{ }

## LOWER\$() String function

**Action:** converts all capital letters of a string expression including all PC umlauts (IBM character set) to lower case letters.

**Syntax:** LOWER\$(a\$)  
*a\$: sexp*

**Abbreviation:** -

**Explanation:** -

**Example:** PRINT LOWER\$("HELLO GFA, ÖÖÖHA")

```
// Prints  
// hello gfa, öööha
```

**Remarks:** -

**See**

**Also:** UPPER\$(), UCASE\$(), LCASE\$(), XLATE\$()

### LPEEK() Function

**Action:** reads a double word (32 bits) from an address.

**Syntax:** LPEEK(addr)  
*addr: address*

**Explanation:** Reads a double word (32 bits) from an address.

**Example:** PRINT LPEEK(\*a) // prints the address of a  
  
PRINT LPEEK(\$40:\$1E) // reads four bytes from  
// offset \$1E in segment \$40

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

LONG{ } and { } are synonymous with LPEEK() and can be used instead.

addr: see the { } function.

**See**

**Also:** PEEK(), BYTE{ }, DPEEK(), WORD{ }, INT{ }, LONG{ }, { }

## LPOKE Command

**Action:** writes a double word (32 bits) to an address.

**Syntax:** LPOKEaddr,m  
*addr: address*  
*m: iexp*

**Explanation:** Writes a double word (32 bits) to an address.

**Example:** -

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

LONG{ } and { } are synonymous with LPOKE() and can be used instead.

addr: see the { } function.

**See**

**Also:** POKE(), BYTE{ } =, DPOKE(), WORD{ } =, INT{ } =, LONG{ } =, { } =,

### LSET String function

**Action:** moves a string expression, left justified, to a string.

**Syntax:** LSET a\$=b\$  
*a\$:* svar  
*b\$:* sexp

**Abbreviation:** -

**Explanation:** LSET a\$=b\$ will, first of all, replace all characters in a\$ with spaces. Next, b\$ is moved into a\$ left justified. If b\$ contains more characters than a\$, then only as many characters as there are "places" for in a\$ are moved.

**Example:**

```
a$=STRING$(15,"-")
b$="Hello GFA
PRINT a$'LEN(a$) // prints ----- 15
PRINT b$'LEN(b$) // prints Hello GFA 9
LSET a$=b$
PRINT a$'LEN(a$) // prints Hello GFA 15
```

**Remarks:** -

**See**

**Also:** RSET, MID\$





### MALLOC() Function

**Action:** reserves memory.

**Syntax:** MALLOC(*n*)  
*n*: *iexp*

**Abbreviation:** -

**Explanation:** MALLOC(*n*) allocates memory to the program. *n* specifies the amount of memory to reserve in bytes. If *n* is -1, MALLOC(-1) returns the size of the largest available block of free memory.

If *n* is positive, MALLOC reserves the amount of memory specified in *n* and returns the address of this memory.

**Example:**  
PRINT MALLOC(-1)  
a%=MALLOC(2000)  
PRINT a%

**Remarks:** -

**See**

**Also:** MFREE(), MSHRINK()

## MAT ADD Command

**Action:** adds all elements in two (one- or two-dimensional) floating point arrays.

**Syntax:** MAT ADD a() $=$ b()+c() or  
MAT ADD a(),b() or  
MAT ADD a(),x

*a(), b(), c(): names of one- or two-dimensional floating point arrays*

*x: aexp*

**Explanation:** MAT ADD a() $=$ b()+c() is only valid for floating point arrays of the same order, such as DIM a(n,m),b(n,m),c(n,m) or DIM a(n),b(n),c(n). The contents of elements in array c() are added to the contents of elements in array b() and the result is written to array a().

MAT ADD a(),b(): adds the contents of elements in array b() to the elements in array a() and writes the result to array a(). The original array a() is thereby lost.

MAT ADD a(),x adds the expression x to the contents of all elements in array a() and writes the result to array a(). The original array a() is thereby lost.

**Example:**

```
DIM a(3,5),b(3,5),c(3,5)
MAT SET b() $=$ 3
MAT SET c() $=$ 4
MAT PRINT b()
PRINT STRING$(10,"-")
MAT PRINT c()
PRINT STRING$(10,"-")
MAT ADD a() $=$ b()+c()
MAT PRINT a()
```

## Commands and functions

---

```
// prints 3,3,3,3,3
// 3,3,3,3,3
// 3,3,3,3,3
// -----
// 4,4,4,4,4
// 4,4,4,4,4
// 4,4,4,4,4
// -----
// 7,7,7,7,7
// 7,7,7,7,7
// 7,7,7,7,7
```

```
DIM a(3,5),b(3,5)
MAT SET a()=1
MAT SET b()=3
MAT PRINT a()
PRINT STRING$(10,"-")
MAT PRINT b()
PRINT STRING$(10,"-")
MAT ADD a(),b()
MAT PRINT a()
// prints 1,1,1,1,1
// 1,1,1,1,1
// 1,1,1,1,1
// -----
// 3,3,3,3,3
// 3,3,3,3,3
// 3,3,3,3,3
// -----
// 4,4,4,4,4
// 4,4,4,4,4
// 4,4,4,4,4
```

```
DIM a(3,5)
MAT SET a()=1
MAT PRINT a()
PRINT STRING$(10,"-")
```

```
MAT ADD a(),5
MAT PRINT a()
// prints 1,1,1,1,1
//      1,1,1,1,1
//      1,1,1,1,1
//      -----
//      6,6,6,6,6
//      6,6,6,6,6
//      6,6,6,6,6
```

**Remarks:** -

**See**

**Also:** MAT SUB, MAT MUL

### MAT BASE Command

**Action:** sets the row and column offsets for array indexing.

**Syntax:** MAT BASE 0 or MAT BASE 1

**Explanation:** The MAT BASE command is only relevant when OPTION BASE 0 is in effect. MAT BASE 1 can then set the offset for start of column and row for indexing of matrix operation on one and two dimensional floating point arrays to 1. MAT BASE 0 sets the offset after MAT BASE 1 back to 0.

**Example:**

```
OPTION BASE 0
MAT BASE 1
DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
DIM a(3,3)
MAT READ a()
PRINT a(1,1)           // prints      1
```

**Remarks:** The setting with MAT BASE n is only valid for commands MAT READ, MAT PRINT, MAT CPY, MAT XCPY, MAT ADD, MAT SUB and MAT MUL. The default is MAT BASE 1.

**See**

**Also:** OPTION BASE

## MAT CLR Command

**Action:** sets all elements in a one- or two-dimensional floating point array to 0.

**Syntax:** MAT CLR a()  
*a(): name of a one- or two-dimensional floating point array*

**Explanation:** MAT CLR a() is equivalent to ARRAYFILL a(),0, that is to say the command sets all elements of array a() to 0.

**Example:**

```
DATA 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
DIM a(3,3)
MAT READ a()
PRINT a(1,1)
MAT CLR a()
PRINT a(1,1)

// First it prints 1, and then 0.
```

**Remarks:** -

**See**

**Also:** ARRAYFILL, MAT SET, MAT ONE, MAT NEG

## MAT CPY Command

**Action:** copies a number of rows with a number of elements, from row/column offset in the source matrix to row/column offset in the target matrix.

**Syntax:**  $\text{MAT CPY } a([i,j]) = b([k,l])[h,w]$   
 $i,j,k,l,w,h:$  *iexp*  
 $a(), b():$  *one or two dimensional floating point arrays*

**Explanation:**  $\text{MAT CPY } a([i,j]) = b([k,l])[h,w]$  copies  $h$  rows with  $w$  elements in matrix  $b()$ , from  $l$  and  $k$  row/column offset in matrix  $b()$  to  $i$  and  $j$  row/column offset in matrix  $a()$ . The maximum number of elements copied is equivalent to the minimum number allowed when dimensioning the matrices, the number of rows ( $h$ ) and the number of elements per row ( $w$ ).

If MAT CPY is used on vectors  $j$  and  $l$  are ignored. Following a DIM  $a(n), b(m)$  the  $a()$  and  $b()$  are interpreted as row vectors, that is to say as matrices of type  $(1,n)$  and  $(1,m)$ .

To handle  $a()$  and  $b()$  as column vectors, they must be dimensioned as matrices of type  $(n,1)$  and  $(m,1)$ , that is to say as DIM  $a(n,1), b(m,1)$ .

MAT CPY always handles vectors as column vectors, regardless of their type, so in order to use the correct MAT CPY syntax with vectors MAT CPY  $a(n,1) = b(m,1)$  must always be used.

If the  $h$  and  $w$  parameters in MAT CPY are given explicitly, the following rules apply when copying vectors:

When  $w = > 1$  only the  $h$  parameter is taken into account. When  $w = 0$  no copying takes place.



When  $h \geq 1$ , the  $w$  is taken into account only when  $b()$  is a row vector and  $a()$  is a column vector. Here too, no copying takes place when  $h=0$ .

**Example:**

```
DIM a(3,5),b(6,6)
MAT BASE 1
MAT SET a()= -1
MAT SET b()=5
MAT CPY a(2,2)=b(3,4),3,3
MAT PRINT a()           // prints    -1,-1,-1,-1,-1
//                      -1,5,5,5,-1
//                      -1,5,5,5,-1
```

**Remarks:**

If some indices are dropped - due to the given width ( $w$ ) or height ( $h$ ) - MAT CPY can result in the following special cases:

**MAT CPY a()=b()**

copies into matrix  $a()$  all elements of matrix  $b()$  for which there are identical indices in matrix  $a()$ .

**Example:**

```
DIM a(3,5),b(6,6)
MAT BASE 1
MAT SET a()= -1
MAT SET b()=5
MAT CPY a()=b()
MAT PRINT a()           // prints    5,5,5,5,5
//                      5,5,5,5,5
//                      5,5,5,5,5
```

### **MAT CPY a(i,j)=b()**

copies all elements in matrix b(), from row/column offset defined with MAT BASE, to row/column offset defined with i and j in matrix a(). The maximum number of elements copied is equivalent to the minimum number allowed when dimensioning the matrices, the number of rows (h) and the number of elements per row (w).

#### **Example:**

```
DIM a(3,5),b(6,6)
MAT BASE 1
MAT SET a()= -1
MAT SET b()=5
MAT CPY a(2,2)=b()
MAT PRINT a()           // prints   -1,-1,-1,-1,-1
//                      -1,5,5,5,5
//                      -1,5,5,5,5
```

### **MAT CPY a()=b(k,l)**

copies all elements in matrix b(), from row/column offset defined with k and l, to row/column offset defined with MAT BASE in matrix a(). The maximum number of elements copied is equivalent to the minimum number allowed when dimensioning the matrices, the number of rows (h) and the number of elements per row (w).

#### **Example:**

```
DIM a(3,5),b(6,6)
MAT BASE 1
MAT SET a()= -1
MAT SET b()=5
MAT CPY a()=b(4,4)
MAT PRINT a()           // prints   5,5,5,-1,-1
//                      5,5,5,-1,-1
//                      5,5,5,-1,-1
```

**MAT CPY a(i,j) = b(k,l)**

copies all elements in matrix b(), from row/column offset defined with k and l, to row/column offset defined with i and j in matrix a(). The maximum number of elements copied is equivalent to the minimum number allowed when dimensioning the matrices, the number of rows (h) and the number of elements per row (w).

**Example:**

```
DIM a(3,5),b(6,6)
MAT BASE 1
MAT SET a()= -1
MAT SET b()=5
MAT CPY a(2,2)=b(4,4)
MAT PRINT a()           // prints   -1,-1,-1,-1,-1
//                       -1,5,5,5,-1
//                       -1,5,5,5,-1
```

**MAT CPY a() = b(),h,w**

copies h rows and w elements in matrix b(), from row/column offset defined with MAT BASE, to row/column offset defined with MAT BASE in matrix a(). The maximum number of elements copied is equivalent to the minimum number allowed when dimensioning the matrices, the number of rows (h) and the number of elements per row (w).

**Example:**

```
DIM a(3,5),b(6,6)
MAT BASE 1
MAT SET a()= -1
```

## Commands and functions

---

```
MAT SET b()=5
MAT CPY a()=b(),3,3
MAT PRINT a()      // prints      5,5,5,-1,-1
//                  5,5,5,-1,-1
//                  5,5,5,-1,-1
```

**See**

**Also:**

**MAT XCPY, MAT TRANS**

## MAT DET Command

**Action:** calculates the determinant of a two-dimensional floating point array which is interpreted as a matrix.

**Syntax:** MAT DET x=a([i,j])[n]  
*a(): name of a two-dimensional floating point array*  
*x: aexp*  
*i,j,n: iexp*

**Explanation:** MAT DET x=a([i,j])[n] calculates the determinant of a square matrix of type (n,n). Assuming that OPTION BASE 1 is in effect, the row and column offsets for MAT BASE 0 and MAT BASE 1 default to a(0,0) and a(1,1) respectively. A determinant of a square section of a matrix can also be calculated. This matrix section is defined by i and j for row and column offsets in a() and by n for the number of elements. An internal matrix of (n,n) type is thereby created at i-th row and j-th column.

**Example:**

```
DATA 2,4.5,6,3.2,7,1.7,-4,12
DATA -3,5,9,-2.1,6,9,11,3
DATA 11.4,2.3,6,3.2,6,1.2,-5,7
DATA 3,5,6,8.2,4.1,-5.2,6.2,7.9
DATA 1,2.3,9,8.1,0,4.2,5,3.7
DATA 4.2,7.1,8.3,9.1,-5,-3,-1,0
DATA 2.0,3,9.1,0,0,7.1,-3,8.8
DATA 2.1,9,3.3,4,5,-1,-2,0
DIM a(8,8),b(4,4)
MAT READ a()
MAT PRINT a(),5,2      // original matrix
PRINT
MAT DET x=a()          // calculate the determinant
PRINT "Determinant = ";x
PRINT
```

## Commands and functions 8.470

---

```
MAT DET y=a(3,2),4    // calculates the determinant
//                  of a matrix segment
PRINT "Segment determinant = ";y
PRINT
PRINT "Test:"
PRINT
PRINT "Matrix segment:"
PRINT
MAT CPY b( )=a(3,2),4,4
MAT PRINT b( ),5,2
PRINT
MAT DET z=b( )
PRINT "Determinant = ";z
```

prints

```
2.00, 4.50, 6.00, 3.20, 7.00, 1.70,-4.00,12.00
-3.00, 5.00, 9.00,-2.10, 6.00, 9.00,11.00, 3.00
11.40, 2.30, 6.00, 3.20, 6.00, 1.20,-5.00, 7.00
3.00, 5.00, 6.00, 8.20, 4.10,-5.20, 6.20, 7.90
1.00, 2.30, 9.00, 8.10, 0.00, 4.20, 5.00, 3.70
4.20, 7.10, 8.30, 9.10,-5.00,-3.00,-1.00, 0.00
2.00, 3.00, 9.10, 0.00, 0.00, 7.10,-3.00, 8.80
2.10, 9.00, 3.30, 4.00, 5.00,-1.00,-2.00, 0.00
```

```
Determinant          = -15045648.8341
Segment determinant = 1337.7709
Test:
```

**Matrix segment:**

```
2.30, 6.00, 3.20, 6.00
5.00, 6.00, 8.20, 4.10
2.30, 9.00, 8.10, 0.00
7.10, 8.30, 9.10,-5.00
```

```
Determinant = 1337.7709
```

**Remarks:**

-

**See**

**Also:**

MAT QDET, MAT RANG, MAT INV

### MAT INPUT # Command

**Action:** reads a file into a floating point array.

**Syntax:** MAT INPUT #i,a()  
*a(): name of a floating point array*  
*i: ivar*

**Abbreviation:** m i #1,a()

**Explanation:** MAT INPUT #1,a() reads the values of a floating point array in ASCII from a file (the format of which is the reverse of MAT PRINT, the commas and line feeds have the same meaning as in INPUT #). The file channel #1 must be opened with OPEN "i",#1... prior to the execution of MAT INPUT #!

**Example:**

```
OPEN "o",#1,"Test.DAT"
DIM a(3,3)
MAT ONE a()
MAT PRINT #1,a()
CLOSE #1
MAT CLR a()
OPEN "i",#1,"Test.DAT"
MAT INPUT #1,a()
CLOSE #1
MAT PRINT a()

// prints 1,0,0
// 0,1,0
// 0,0,1
```

**Remarks:** -

**See**

**Also:** MAT READ, MAT PRINT



## MAT INV Command

**Action:** calculates an inverse of a two-dimensional floating point array which is interpreted as a matrix.

**Syntax:** MAT INV a()=b()  
*a(),b(): names of two-dimensional floating point arrays with the same number of rows and columns.*

**Explanation:** MAT INV a()=b() returns the inverse of a square matrix. The inverse of matrix b() is written to matrix a(). a() must, therefore, be of the same type as b().

**Example:**

```
DATA 2,4.5,6,3.2,7,1.7,-4,12
DATA -3,5,9,-2.1,6,9,11,3
DATA 11.4,2.3,6,3.2,6,1.2,-5,7
DATA 3,5,6,8.2,4.1,-5.2,6.2,7.9
DATA 1,2.3,9,8.1,0,4.2,5,3.7
DATA 4.2,7.1,8.3,9.1,-5,-3,-1,0
DATA 2.0,3,9.1,0,0,7.1,-3,8.8
DATA 2.1,9,3.3,4,5,-1,-2,0
DIM a(8,8),b(8,8),c(8,8),d(8,8)
MAT READ b()
PRINT "Original matrix:
PRINT
MAT PRINT b(),5,2
PRINT
MAT INV a()=b()      // calculate the inverse
PRINT
PRINT "Inverse of b()" :
PRINT
MAT PRINT a(),6,3
REPEAT
  a$=INKEY$
UNTIL a$=CHR$(27)
CLS
```

## Commands and functions

---

```
PRINT "Inverse * Original matrix "  
PRINT  
MAT MUL c()=a()*b()  
MAT PRINT c(),6,3  
PRINT  
PRINT "Original matrix * Inverse "  
PRINT  
MAT MUL d()=b()*a()  
MAT PRINT d(),6,3
```

prints

### Original matrix:

```
2.00, 4.50, 6.00, 3.20, 7.00, 1.70,-4.00,12.00  
-3.00, 5.00, 9.00,-2.10, 6.00, 9.00,11.00, 3.00  
11.40, 2.30, 6.00, 3.20, 6.00, 1.20,-5.00, 7.00  
3.00, 5.00, 6.00, 8.20, 4.10,-5.20, 6.20, 7.90  
1.00, 2.30, 9.00, 8.10, 0.00, 4.20, 5.00, 3.70  
4.20, 7.10, 8.30, 9.10,-5.00,-3.00,-1.00, 0.00  
2.00, 3.00, 9.10, 0.00, 0.00, 7.10,-3.00, 8.80  
2.10, 9.00, 3.30, 4.00, 5.00,-1.00,-2.00, 0.00
```

### Inverse of b():

```
-0.337,-0.127, 0.058, 0.182, 0.041,-0.215, 0.276, 0.191  
-0.400,-0.183,-0.083, 0.252, 0.037,-0.297, 0.432, 0.375  
0.648, 0.378, 0.100,-0.401,-0.201, 0.605,-0.647,-0.547  
-0.098,-0.129,-0.033, 0.052, 0.183,-0.146, 0.080, 0.142  
0.331, 0.166, 0.068,-0.199,-0.041, 0.195,-0.366,-0.186  
-0.387,-0.208,-0.044, 0.167, 0.198,-0.371, 0.401, 0.336  
-0.259,-0.072,-0.020, 0.186, 0.043,-0.184, 0.209, 0.136  
-0.233,-0.156,-0.060, 0.216, 0.041,-0.239, 0.321, 0.169
```

### Inverse \* Original matrix =

```
1.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000,-0.000
0.000, 1.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000
-0.000,-0.000, 1.000,-0.000,-0.000,-0.000,-0.000,-0.000,-0.000
0.000, 0.000, 0.000, 1.000, 0.000,-0.000, 0.000,-0.000
-0.000,-0.000,-0.000, 0.000, 1.000,-0.000,-0.000,-0.000,-0.000
0.000, 0.000, 0.000, 0.000, 0.000, 1.000, 0.000, 0.000, 0.000
0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 1.000, 0.000, 0.000
0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 1.000, 0.000
0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 0.000, 1.000
```

### Original matrix \* Inverse =

```
1.000,-0.000,-0.000, 0.000, 0.000,-0.000, 0.000, 0.000, 0.000
0.000, 1.000, 0.000,-0.000, 0.000, 0.000,-0.000, 0.000, 0.000
0.000, 0.000, 1.000,-0.000, 0.000, 0.000,-0.000, 0.000, 0.000
-0.000,-0.000,-0.000, 1.000, 0.000,-0.000, 0.000, 0.000, 0.000
0.000, 0.000, 0.000,-0.000, 1.000, 0.000,-0.000, 0.000, 0.000
-0.000, 0.000, 0.000, 0.000,-0.000, 1.000,-0.000, 0.000, 0.000
-0.000,-0.000, 0.000,-0.000,-0.000,-0.000, 1.000, 0.000, 0.000
0.000, 0.000, 0.000,-0.000,-0.000, 0.000,-0.000, 1.000, 0.000
0.000, 0.000, 0.000,-0.000,-0.000, 0.000,-0.000, 0.000, 1.000
```

**Remarks:** -

**See**

**Also:** MAT DET, MAT QDET, MAT RANK

## MAT MUL Command

**Action:** multiplies one- or two-dimensional floating point arrays which are interpreted as matrices.

**Syntax:**

```
MAT MUL a()=b()*c()    or
MAT MUL x=a()*b()      or
MAT MUL x=a()*b()*c()  or
MAT MUL a(),x
a(),b(),c(): names of one- or two-dimensional
              floating point arrays
x:           aexp
```

**Explanation:** MAT MUL a()=b()\*c() is intended for 'related' matrices of the same order. Matrices b() and c() are multiplied. The result of this multiplication is written to matrix a(). In order to get a product of a matrix multiplication, the matrix on the left (in this case matrix b()) must have the same number of columns as the matrix on the right (in this case c()) has rows.

The matrix a() must, in this example, have the same number of rows as b() and the same number of columns as c(), i.e. DIM a(2,2),b(2,3),c(3,2)

Matrices are multiplied using the formula 'rows times columns'. I.e. the elements in a(i,j) are obtained by multiplying the elements of the i-th row in matrix b() with the j-th column in matrix c() and the individual products are added up. If vectors are used instead of matrices, MAT MUL a()=b()\*c() produces the dyadic product of two vectors.

MAT MUL x=a()\*b() is intended for vectors with the same number of elements. The result x is the scalar product of vectors a() and b(). The scalar product of two vectors is defined as the sum of n products a(i)\*b(i), i=1,...,n.

`MAT MUL x=a()*b()*c()` is intended for 'related' vectors `a()` and `c()` as well as 'related' matrix `b()`. The result is scalar `x`, which is obtained when the (row-) vector `a()` is multiplied with matrix `b()` and this resulting vector is then multiplied with (column-) vector `c()`. The vector `a()` must thereby have as many elements as the matrix `b()` has rows. The vector `c()` must contain the same number of elements as the matrix `b()` has columns, for example `DIM a(5),b(5,3),c(3)`.

`MAT MUL a(),x` multiplies the matrix or vector `a()` with the expression `x`.

#### Example:

```

DIM a(2,2),b(2,3),c(3,2) // MAT MUL a()=b()*c()
MAT SET b()=1
DATA 1,2,-3,4,5,-1
MAT READ c()
MAT PRINT b(),5,1
PRINT STRING$(18,"-")
MAT PRINT c(),5,1
PRINT STRING$(18,"-")
MAT MUL a()=b()*c()
MAT PRINT a(),5,1
//
// prints 1.0, 1.0, 1.0
// 1.0, 1.0, 1.0
// -----
// 1.0, 2.0
// -3.0, 4.0
// 5.0, -1.0
// -----
// 3.0, 5.0
// 3.0, 5.0

DIM a(3,3),b(3),c(3) // MAT MUL a()=b()*c()
DATA 1,2,-3,4,5,-1
MAT READ b()
MAT READ c()
MAT PRINT b(),5,1

```

## Commands and functions

---

```
PRINT STRING$(18,"-")
MAT PRINT c(),5,1
PRINT STRING$(18,"-")
MAT MUL a()=b()*c()
MAT PRINT a(),5,1
// prints 1.0, 2.0, -3.0
// -----
// 4.0, 5.0, -1.0
// -----
// 4.0, 5.0, -1.0
// 8.0, 10.0, -2.0
// -12.0,-15.0, 3.0
```

```
DIM b(3),c(3) // MAT MUL x=a()*b()
DATA 1,2,-3,4,5,-1
MAT READ b()
MAT READ c()
MAT PRINT b(),5,1
PRINT STRING$(18,"-")
MAT PRINT c(),5,1
PRINT STRING$(18,"-")
MAT MUL x=b()*c()
PRINT x
// prints 1.0, 2.0, -3.0
// -----
// 4.0, 5.0, -1.0
// -----
// 17.0
```

```
DIM a(2),b(2,3),c(3) // MAT MUL x=a()*b()*c()
DATA 1,2,-3,4,5
MAT READ a()
MAT READ c()
MAT SET b()=1
MAT PRINT a(),5,1
PRINT STRING$(18,"-")
MAT PRINT b(),5,1
```

```

PRINT STRING$(18,"-")
MAT PRINT c(),5,1
PRINT STRING$(18,"-")
MAT MUL x=a()*b()*c()
PRINT x
// prints 1.0, 2.0
// -----
// 1.0, 1.0, 1.0
// 1.0, 1.0, 1.0
// -----
// -3.0, 4.0, 5.0
// -----
// 18.0

```

**Remarks:** -

**See**

**Also:** MAT ADD, MAT SUB

### MAT NEG Command

**Action:** negates all elements in a one- or two-dimensional floating point array

**Syntax:** MAT NEG a()  
*a(): name of a one- or two-dimensional floating point array*

**Explanation:** MAT NEG a() multiplies all elements of a one or two dimensional floating point array a() with -1.

**Example:**

```
DIM a(3,3)
MAT ONE a()
MAT PRINT a()
PRINT
MAT NEG a()
MAT PRINT a()
//
// prints first
// 1,0,0
// 0,1,0
// 0,0,1
// and then
// -1,0,0
// 0,-1,0
// 0,0,-1
```

**Remarks:** -

**See**

**Also:** MAT CLR, MAT SET, MAT ONE



## MAT NORM Command

**Action:** row- or column-wise normalising of a two-dimensional floating point array which is interpreted as a matrix.

**Syntax:** MAT NORM a(),i  
*a(): name of a two-dimensional floating point array*  
*i: ivar; i=0 for row-wise and i=1 for column-wise normalising*

**Explanation:** MAT NORM a(),0 and MAT NORM a(),1 are used for both matrices and vectors. MAT NORM a(),0 normalises a matrix (or a vector) row-wise and MAT NORM a(),1 normalises a matrix (or a vector) column-wise. This means that in case of row-wise (column-wise) normalising the sum of squares of all elements in each row (column) is equal to 1.

**Example:**

```
n%=8
DIM a(n%,n%),b(n%,n%),v(n%)
DATA 1,2,3,4,5,6,7,8
DATA 3.2,4,-5,2.4,5.1,6.2,7.2,8.1
DATA -2,-5,-6,-1.2,-1.5,-6.7,4.5,8.1
DATA 5,-2.3,4,5.6,12.2,18.2,14.1,16
DATA 4.1,5.2,16.7,18.4,19.1,20.2,13.6,14.8
DATA 15.2,-1.8,13.6,-4.9,5.4,19.8,16.4,-20.9
DATA -3.6,6,-8.2,-9.1,4,-2.5,2,3.4
DATA 4.7,8.3,9.4,10.5,11,19,15.4,18.9
//
MAT READ a()
MAT CPY b()=a()           // save the original matrix
PRINT "Original matrix"
PRINT
MAT PRINT a(),7,2
REPEAT
  a$=INKEY$
UNTIL a$=CHR$(27)
```

## Commands and functions

---

```
//  
// row-wise normalising  
//  
CLS  
MAT NORM a(),0  
PRINT  
PRINT "Row-wise normalised: "  
PRINT  
MAT PRINT a(),7,2  
REPEAT  
  a$=INKEY$  
UNTIL a$=CHR$(27)  
//  
// testing of the row-wise normalising  
//  
PRINT  
PRINT "Test: "  
PRINT  
FOR i%=1 TO n%  
  MAT XCPY v()=a(i%,1) // copies a() row-wise into  
  // vector v()  
  MAT MUL x=v()*v() // calculates the scalar  
  // product of v() and v()  
  PRINT x'  
NEXT i%  
PRINT  
PRINT  
REPEAT  
  a$=INKEY$  
UNTIL a$=CHR$(27)  
//
```

```
// column-wise normalising
//
CLS
MAT CPY a()=b()      // copy the original matrix
//                  back
MAT NORM a(),1
PRINT "Column-wise normalised: "
PRINT
MAT PRINT a(),7,2
REPEAT
  a$=INKEY$
UNTIL a$=CHR$(27)
//
// testing of column-wise normalising
//
PRINT
PRINT "Test: "
PRINT
FOR i%=1 TO n%
  MAT CPY v()=a(1,i%) // copies a() column-wise
  //                  into vector v()
  MAT MUL x=v()*v()    // calculates the scalar
  //                  product of v() and v()
  PRINT x'
NEXT i%
```

prints

### Original matrix

1.00,	2.00,	3.00,	4.00,	5.00,	6.00,	7.00,	8.00
3.20,	4.00,	-5.00,	2.40,	5.10,	6.20,	7.20,	8.10
-2.00,	-5.00,	-6.00,	-1.20,	-1.50,	-6.70,	4.50,	8.10
5.00,	-2.30,	4.00,	5.60,	12.20,	18.20,	14.10,	16.00
4.10,	5.20,	16.70,	18.40,	19.10,	20.20,	13.60,	14.80
15.20,	-1.80,	13.60,	-4.90,	5.40,	19.80,	16.40,	-20.90
-3.60,	6.00,	-8.20,	-9.10,	4.00,	-2.50,	2.00,	3.40
4.70,	8.30,	9.40,	10.50,	11.00,	19.00,	15.40,	18.90

## Commands and functions

---

### Row-wise normalised:

0.07,	0.14,	0.21,	0.28,	0.35,	0.42,	0.49,	0.56
0.21,	0.26,	-0.32,	0.16,	0.33,	0.40,	0.47,	0.52
-0.14,	-0.35,	-0.42,	-0.08,	-0.11,	-0.47,	0.32,	0.57
0.16,	-0.07,	0.13,	0.18,	0.38,	0.57,	0.44,	0.50
0.10,	0.12,	0.39,	0.43,	0.45,	0.47,	0.32,	0.35
0.38,	-0.05,	0.34,	-0.12,	0.14,	0.50,	0.41,	-0.53
-0.23,	0.39,	-0.53,	-0.59,	0.26,	-0.16,	0.13,	0.22
0.13,	0.22,	0.25,	0.28,	0.30,	0.51,	0.42,	0.51

### Test:

1 1 1 1 1 1 1 1

### Column-wise normalised:

0.06,	0.15,	0.11,	0.16,	0.18,	0.15,	0.22,	0.21
0.18,	0.29,	-0.19,	0.10,	0.19,	0.15,	0.23,	0.21
-0.11,	-0.37,	-0.23,	-0.05,	-0.06,	-0.17,	0.14,	0.21
0.28,	-0.17,	0.15,	0.23,	0.45,	0.45,	0.44,	0.42
0.23,	0.38,	0.63,	0.74,	0.71,	0.50,	0.43,	0.39
0.85,	-0.13,	0.51,	-0.20,	0.20,	0.49,	0.51,	-0.54
-0.20,	0.44,	-0.31,	-0.37,	0.15,	-0.06,	0.06,	0.09
0.26,	0.61,	0.35,	0.43,	0.41,	0.47,	0.48,	0.49

### Test:

1 1 1 1 1 1 1 1

Remarks:

-

See

Also:

-

## MAT ONE Command

**Action:** creates a uniform matrix.

**Syntax:** MAT ONE a()  
*a(): name of a two-dimensional floating point array  
with the same number of rows and columns*

**Explanation:** MAT ONE a() creates, from a two dimensional floating point array a() with the same number of rows and columns, an array in which the elements a(1,1), a(2,2), ...,a(n,n) are equal to 1 and all other elements are equal to 0.

**Example:**

```
DIM a(3,3)
MAT ONE a()
MAT PRINT a()
//
//                prints      1,0,0
//                0,1,0
//                0,0,1
```

**Remarks:** -

**See**

**Also:** ARRAYFILL, MAT CLR, MAT SET, MAT NEG

## MAT PRINT Command

**Action:** prints the elements of an array to screen or a channel.

**Syntax:** MAT PRINT [#i,]a()[,g,n]  
*a()*: name of a floating point array  
*i*: ivar  
*g,n*: iexp

**Explanation:** MAT PRINT [#i,]a()[,g,n] prints a floating point array to screen. One-dimensional floating point arrays are printed on one line with individual elements separated by commas. For two-dimensional arrays a line feed is performed after each row. Similar to the PRINT command, the output can optionally be redirected with #i. g and n cause the formatting of the numbers similar to STR\$(x,g,n).

**Example:**

```
DATA 1,2.33333,3
DATA 7,5.25873,9.376
DATA 3.23,7.2,8.999
DIM a(3,3)
MAT READ a()
MAT PRINT a()
PRINT"-----"
MAT PRINT a(),7,3
// prints 1,2.33333,3
// 7,5.25873,9.376
// 3.23,7.2,8.999
// -----
// 1.000,2.333,3.000
// 7.000,5.259,9.376
// 3.230,7.200,8.999
```

**Remarks:** -

**See**

**Also:** MAT READ, MAT INPUT

## MAT QDET Command

**Action:** calculates the determinant of a two-dimensional floating point array which is interpreted as a matrix.

**Syntax:** `MAT QDET x=a([i,j])[,n]`  
*a()*: name of a two dimensional floating point array  
*x*: *axp*  
*i,j,n*: *iexp*

**Explanation:** `MAT QDET x=a([i,j])[,n]` is equivalent to `MAT DET x = a([i,j])[,n]` except that it's optimised for speed not accuracy. As a rule both methods deliver the same result. However, `MAT DET` should always be used in case of 'critical' matrices whose determinant is close to 0.

**Example:**

```
DATA 2,4.5,6,3.2,7,1.7,-4,12
DATA -3,5,9,-2.1,6,9,11,3
DATA 11.4,2.3,6,3.2,6,1.2,-5,7
DATA 3,5,6,8.2,4.1,-5.2,6.2,7.9
DATA 1,2.3,9,8.1,0,4.2,5,3.7
DATA 4.2,7.1,8.3,9.1,-5,-3,-1,0
DATA 2.0,3,9.1,0,0,7.1,-3,8.8
DATA 2.1,9,3.3,4,5,-1,-2,0
DIM a(8,8)
MAT READ a()
MAT PRINT a(),5,2      // original matrix
PRINT
MAT DET x=a()          // calculate the determinant
PRINT "Determinant with MAT DET = ";x
PRINT
MAT QDET y=a()         // calculate the determinant
PRINT "Determinant with MAT QDET = ";y
PRINT
PRINT "Deviation = ";x-y
```



prints

```
2.00, 4.50, 6.00, 3.20, 7.00, 1.70,-4.00,12.00
-3.00, 5.00, 9.00,-2.10, 6.00, 9.00,11.00, 3.00
11.40, 2.30, 6.00, 3.20, 6.00, 1.20,-5.00, 7.00
3.00, 5.00, 6.00, 8.20, 4.10,-5.20, 6.20, 7.90
1.00, 2.30, 9.00, 8.10, 0.00, 4.20, 5.00, 3.70
4.20, 7.10, 8.30, 9.10,-5.00,-3.00,-1.00, 0.00
2.00, 3.00, 9.10, 0.00, 0.00, 7.10,-3.00, 8.80
2.10, 9.00, 3.30, 4.00, 5.00,-1.00,-2.00, 0.00
```

Determinant with MAT DET = -15045648.8341

Determinant with MAT QDET = -15045648.8341

Deviation = 0

**Remarks:**

-

**See**

**Also:**

MAT DET, MAT RANK, MAT INV

## MAT RANK Command

**Action:** returns the rank of a two-dimensional floating point array which is interpreted as a matrix.

**Syntax:** MAT RANK  $x = a([i,j])[n]$   
*a(): name of a two-dimensional floating point array*  
*x: aexp*  
*i,j,n: iexp*

**Explanation:** MAT RANK  $x = a([i,j])[n]$  prints the rank of a square matrix. Analogous to MAT DET and MAT QDET an arbitrary row and column offset can be specified.

To process a section of a matrix, a number of elements is specified in n. An internal matrix of (n,n) type is thereby created at i-th row and j-th column.

**Example:**

```
DATA 2,4,5,6,3.2,7,1.7,-4,12
DATA -3,5,9,-2.1,6,9,11,3
DATA 11.4,2.3,6,3.2,6,1.2,-5,7
DATA 3,5,6,8.2,4.1,-5.2,6.2,7.9
DATA 1,2,3,9,8.1,0,4.2,5,3.7
DATA 4.2,7.1,8.3,9.1,-5,-3,-1,0
DATA 2.0,3,9.1,0,0,7.1,-3,8.8
DATA 2.1,9,3.3,4,5,-1,-2,0
DIM a(8,8)
MAT READ a()
PRINT "Original matrix:
PRINT
MAT PRINT a(),5,2      // original matrix
PRINT
MAT RANK x=a()         // calculate the rank
PRINT "Rank = ";x
```

prints

**Original matrix:**

```
2.00, 4.50, 6.00, 3.20, 7.00, 1.70,-4.00,12.00
-3.00, 5.00, 9.00,-2.10, 6.00, 9.00,11.00, 3.00
11.40, 2.30, 6.00, 3.20, 6.00, 1.20,-5.00, 7.00
3.00, 5.00, 6.00, 8.20, 4.10,-5.20, 6.20, 7.90
1.00, 2.30, 9.00, 8.10, 0.00, 4.20, 5.00, 3.70
4.20, 7.10, 8.30, 9.10,-5.00,-3.00,-1.00, 0.00
2.00, 3.00, 9.10, 0.00, 0.00, 7.10,-3.00, 8.80
2.10, 9.00, 3.30, 4.00, 5.00,-1.00,-2.00, 0.00
```

Rank = 8

**Remarks:**

-

**See**

**Also:**

MAT DET, MAT QDET, MAT INV

### MAT READ Command

**Action:** reads values from DATA lines into a floating point array.

**Syntax:** MAT READ a()  
*a(): name of a floating point array*

**Explanation:** -

**Example:** DATA 1,2,3,4,5,6,7,8,9,10  
DIM a(2,5)  
MAT READ a()  
PRINT a(2,4) // prints a 9

**Remarks:** -

**See**

**Also:** MAT PRINT, MAT INPUT

## MAT SET Command

**Action:** assigns a value to all elements of a one- or two-dimensional floating point array.

**Syntax:** `MAT SET a()=x`  
*a(): name of a one- or two-dimensional floating point array*  
*x: aexp*

**Explanation:** `MAT SET a()=x` is equivalent to an `ARRAYFILL a(),x`, i.e. the command sets all elements of the array `a()` to value `x`.

**Example:**

```
DIM a(5,7)
FOR i%=1 TO 5
  FOR j%=1 TO 7
    a(i%,j%)=RAND(10)
  NEXT j%
NEXT i%
MAT SET a(),5.3
FOR i%=1 TO 5
  FOR j%=1 TO 7
    PRINT a(i%,j%)
  NEXT j%
NEXT i%           // prints the value 5.3 35
                  times.
```

**Remarks:** -

**See**

**Also:** `ARRAYFILL`, `MAT CLR`, `MAT ONE`, `MAT NEG`

## MAT SUB Command

**Action:** subtracts all elements in two (one- or two-dimensional) floating point arrays.

**Syntax:** MAT SUB a() $=$ b()-c() or  
 MAT SUB a(),b() or  
 MAT SUB a(),x  
*a(), b(), c(): names of one- or two-dimensional floating point arrays*  
*x: aexp*

**Explanation:** MAT SUB a() $=$ b()-c() is only valid for floating point arrays of the same order, such as DIM a(n,m),b(n,m),c(n,m) or DIM a(n),b(n),c(n). The contents of elements in array c() are subtracted from the contents of elements in array b() and the result is written to array a().

MAT SUB a(),b() subtracts the contents of elements in array b() from the elements in array a() and writes the result to array a(). The original array a() is thereby lost.

MAT SUB a(),x subtracts the expression x from the contents of all elements in array a() and writes the result to array a(). The original array a() is thereby lost.

**Example:** DIM a(3,5),b(3,5),c(3,5)  
 MAT SET b() $=$ 3  
 MAT SET c() $=$ 4  
 MAT PRINT b()  
 PRINT STRING\$(10,"-")  
 MAT PRINT c()  
 PRINT STRING\$(10,"-")  
 MAT SUB a() $=$ b()-c()  
 MAT PRINT a()

```
//          prints 3,3,3,3,3
//          3,3,3,3,3
//          3,3,3,3,3
//          -----
//          4,4,4,4,4
//          4,4,4,4,4
//          4,4,4,4,4
//          -----
//          -1,-1,-1,-1,-1
//          -1,-1,-1,-1,-1
//          -1,-1,-1,-1,-1
```

```
DIM a(3,5),b(3,5)
MAT SET a()=1
MAT SET b()=3
MAT PRINT a()
PRINT STRING$(10,"-")
MAT PRINT b()
PRINT STRING$(10,"-")
MAT SUB a(),b()
MAT PRINT a()
```

```
//          prints 1,1,1,1,1
//          1,1,1,1,1
//          1,1,1,1,1
//          -----
//          3,3,3,3,3
//          3,3,3,3,3
//          3,3,3,3,3
//          -----
//          -2,-2,-2,-2,-2
//          -2,-2,-2,-2,-2
//          -2,-2,-2,-2,-2
```

## Commands and functions

---

```
DIM a(3,5)
MAT SET a()=1
MAT PRINT a()
PRINT STRING$(10,"-")
MAT SUB a(),5
MAT PRINT a()
//                               prints 1,1,1,1,1
//                               1,1,1,1,1
//                               1,1,1,1,1
//                               -----
//                               -4,-4,-4,-4,-4
//                               -4,-4,-4,-4,-4
//                               -4,-4,-4,-4,-4
```

**Remarks:**

-

**See**

**Also:**

MAT ADD, MAT MUL



## MAT TRANS Command

**Action:** copies a transposed source matrix into a target matrix.

**Syntax:** MAT TRANS a()=b()  
a(),b(): *one- or two-dimensional floating point array*

**Explanation:** MAT TRANS a()=b() copies the transposed matrix b() into matrix a(), assuming that both a() and b() are dimensioned appropriately, that is to say the number of rows in a() must correspond to the number of columns in b(), and the number of columns in a() must correspond to the number of rows in b() (for example DIM a(n,m),b(m,n)).

**Example:**

```
DIM a(4,3),b(3,4)
MAT BASE 1
MAT SET a()=-1
MAT SET b()=5
MAT TRANS a()=b()
MAT PRINT a()           // prints    5,5,5
//                      5,5,5
//                      5,5,5
//                      5,5,5
```

**Remarks:** Defines a square matrix, that is to say, a matrix with the same number of rows and columns so that MAT TRANS a() can be used. This command swaps the rows and columns in matrix a() and writes the modified matrix back to a(). The original matrix a() is thereby lost. (However, it can be restored by performing MAT TRANS a() again.)

**See**

**Also:** MAT CPY, MAT XCPY

## MAT XCPY Command

**Action:** copies a specified number of rows containing a specified number of elements, from the given row/column offset in source matrix to the given row/column offset in target matrix. The source matrix, or the relevant part of it, are internally transposed before copying.

**Syntax:**  $\text{MAT XCPY } a([i,j]) = b([k,l])[h,w]$   
 $i,j,k,l,w,h:$  *iexp*  
 $a(),b():$  *one- or two-dimensional floating point array*

**Explanation:**  $\text{MAT XCPY } a([i,j]) = b([k,l])[h,w]$  copies  $h$  rows with  $w$  elements, from row/column offset defined with  $l$  and  $k$  in matrix  $b()$ , to row/column offset defined with  $i$  and  $j$  in matrix  $a()$ . The maximum number of elements copied is equivalent to the minimum number allowed when dimensioning the matrices, the number of rows ( $h$ ) and the number of elements per row ( $w$ ). The matrix  $b()$ , or the relevant part of it, are internally transposed before copying, that is to say the rows and column are swapped. This change affects only the copy and not the matrix  $b()$  itself.

If MAT XCPY is used on vectors  $j$  and  $l$  are ignored. Following a DIM  $a(n),b(m)$  the  $a()$  and  $b()$  are interpreted as row vectors, that is to say as matrices of type  $(1,n)$  and  $(1,m)$ .

To handle  $a()$  and  $b()$  as column vectors, they must be dimension as matrices of type  $(n,1)$  and  $(m,1)$ , that is to say as DIM  $a(n,1),b(m,1)$ .

If both vectors are of the same type, that is to say they are both rows or columns, MAT CPY must be used.

If the *h* and *w* parameters in MAT XCPY are given explicitly, the following rules apply when copying vectors:

When *w* = > 1, the *h* parameter is taken into account only when *b()* is a column vector and *a()* is a row vector. When *w*=0 no copying takes place.

When *h* = > 1 the *w* parameter is taken into account only when *b()* is a row vector and *a()* is a column vector. When *h*=0 no copying takes place.

**Example:**

```
DIM a(3,5),b(7,2)
MAT BASE 1
MAT SET a()=-1
MAT SET b()=5
MAT XCPY a(1,2)=b(3,2)
MAT PRINT a()           // prints      1,5,5,5,5
//                      1,-1,-1,-1,-1
//                      1,-1,-1,-1,-1
```

**Explanation:**

If some indices are dropped - due to the given width (*w*) or height (*h*) - the following special cases can result just like with MAT CPY:

```
MAT XCPY a()=b()
MAT XCPY a([i,j])=b()
MAT XCPY a()=b([k,l])
MAT XCPY a()=b(),w,h
```

These act the same as the corresponding MAT CPY commands, except for the transposition of relevant areas of matrix *b()* before copying to matrix *a()*. The *b()* matrix remains unchanged!

**See**

**Also:**

MAT CPY, MAT TRANS

### MAX() Function

**Action:** returns the largest argument.

**Syntax:** MAX(x1,x2[,x3,...,xn]) or  
MAX(x1\$,x2\$[,x3\$,...,xn\$])  
*x1,x2,...: aexp*  
*x1\$,x2\$,...: sexp*

**Explanation:** MAX(x1,x2[,x3,...,xn]) determines the largest of numeric expressions x1,..xn. In case of strings each single ASCII code in a string is compared character by character.

**Example:** PRINT MAX(2,PI,SQR(64)) // prints 8  
PRINT MAX("ABBBBBBBBB","B") // prints B

**Remarks:** -

**See**

**Also:** MIN()

## MEMAND Command

**Action:** performs a logical bit-wise AND of two bit patterns in memory.

**Syntax:** MEMAND *src\_addr*,*dst\_addr*,*count*  
*src\_addr*,*dst\_addr*: *address*  
*count*: *iexp*

**Abbreviation:** -

**Explanation:** MEMAND *src\_addr*, *dst\_addr*, *count* performs a logical AND of two memory locations.

*src\_addr* specifies the address of the source and *dst\_addr* the address of the destination location. *count* specifies the number of bytes to use at both locations. The logical AND results in the target bits being set only when the corresponding bits are set in both source and target area.

**Example:**

```
SCREEN 3
DIM a!(15),b!(15)
a!(9)=-1,b!(9)=-1,b!(10)=-1,b!(12)=-1
SCREEN 3
@test
MEMAND V:a!(0),V:b!(0),(DIM?(a!())+7)>>3
@test
REPEAT
UNTIL LEN(INKEY$)
//
PROCEDURE test
  FOR i%=0 TO 15
    PRINT STR$(a!(i%),2,0)'
  NEXT i%
  PRINT
  FOR i%=0 TO 15
```

## Commands and functions

---

```
        PRINT STR$(b!(i%),2,0)'
NEXT i%
PRINT
RETURN

// Performs a bit-wise AND of arrays a!() and b!().
// The program prints
//
// 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 -1 -1 0 -1 0 0 0
// 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0
//
// This means that after the AND operation only the
// element 9 in array b!()=TRUE.
```

### Remarks:

This method of memory manipulation can be particularly handy for use with data bases. For example, if a data base contains variables which are used as flags to mark (-1) or not to mark (0) an attribute, these method of memory manipulation can be very helpful. Using MEMAND an inquiry can be made to see if the markers apply to one or both attributes.

addr: see the {} function.

### See

### Also:

MEMOR, MEMXOR

## MEMBFILL Command

**Action:** fills memory area with a byte value.

**Syntax:** MEMBFILL addr,count,value  
*addr:*           *address*  
*count,value:* *iexp*

**Abbreviation:** -

**Explanation:** By using MEMBFILL addr,count,value it is possible to fill an area directly with a one byte value. addr specifies the start address of the memory area, count the number of bytes to fill and value the byte expression.

**Example:**

```
SCREEN 3
MEMBFILL $B800:0,100,97
REPEAT
UNTIL LEN(INKEY$)

// Writes a lowercase a 50 times on the top line
// of the screen (EGA). The character attribute is 97
// ($61), blue on orange.
```

**Remarks:** When used on arrays MEMBFILL can have the same effect as ARRAYFILL.  
addr: see the {} function.

**See**

**Also:** MEMWFILL, MEMLFILL, ARRAYFILL

### MEMLFILL Command

**Action:** fills memory area with a longword (4-byte) value.

**Syntax:** MEMLFILL *addr*,*count*,*value*  
*addr:* *address*  
*count,value:* *iexp*

**Abbreviation:** -

**Explanation:** By using MEMLFILL *addr*,*count*,*value* it is possible to fill an area directly with a double word (4-byte) value. *addr* specifies the start address of the memory area, *count* the number of bytes to fill and *value* the double word expression.

**Example:**

```
SCREEN 3
DIM a%(10)
MEMLFILL V:a%(0),DIM?(a%())*4,100000
FOR i%=0 TO 10
    PRINT a%(i%)
NEXT i%
REPEAT
UNTIL LEN(INKEY$)
```

```
// Dimensions a word array with eleven elements and
// fills them with the value 100000. The values in
// the array are then printed on the screen.
```

**Remarks:** ARRAYFILL is much better suited for filling of arrays.

*addr:* see the {} function.

**See**

**Also:** MEMBFILL, MEMWFILL, ARRAYFILL



## MEMOR Command

**Action:** performs a logical bit-wise OR of two bit patterns in memory.

**Syntax:** MEMOR *addr*  
*addr:*        *address*  
*count:*       *iexp*

**Abbreviation:** -

**Explanation:** MEMOR *src\_addr*, *dst\_addr*, *count* performs a logical OR of two memory locations.

*src\_addr* specifies the address of the source and *dst\_addr* the address of the destination location. *count* specifies the number of bytes to use at both locations. The logical OR results in the target bits being set when either the source or the target bits are also set.

**Example:**

```
SCREEN 3
DIM a!(15),b!(15)
a!(9)=-1,b!(10)=-1,b!(12)=-1
SCREEN 3
@test
MEMOR V:a!(0),V:b!(0),(DIM?(a!())+7)>>3
@test
REPEAT
UNTIL LEN(INKEY$)
//
PROCEDURE test
  FOR i%=0 TO 15
    PRINT STR$(a!(i%),2,0)'
  NEXT i%
  PRINT
  FOR i%=0 TO 15
    PRINT STR$(b!(i%),2,0)'
```

## Commands and functions

---

```
NEXT i%
PRINT
RETURN

// performs a bit-wise OR of arrays a1() and b1().
// The program prints
//
// 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 -1 0 -1 0 0 0
// 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 -1 -1 0 -1 0 0 0

// This means that after the OR operation only
// the elements 9, 10 and 12 in array b1()=TRUE.
```

### Remarks:

This method of memory manipulation can be particularly handy for use with data bases. For example, if a data base contains variables which are used as flags to mark (-1) or not to mark (0) an attribute, this method of memory manipulation can be very helpful. Using MEMOR an inquiry can be made to see if the markers apply to one, the other or both attributes.

addr: see the {} function.

### See

### Also:

MEMXOR, MEMAND

## MEMWFILL Command

**Action:** fills memory area with a word (2-byte) value.

**Syntax:** MEMWFILL *addr*,*count*,*value*  
*addr:*           *address*  
*count,value:* *iexp*

**Abbreviation:** -

**Explanation:** By using MEMWFILL *addr*,*count*,*value* it is possible to fill an area directly with a word (2-byte) value. *addr* specifies the start address of the memory area, *count* the number of bytes to fill and *value* the word expression.

**Example:** SCREEN 3  
MEMWFILL\$B800:0,400,\$1e03  
  
// Fills an EGA text screen with yellow (\$E) hearts  
// (\$03) on blue (\$1) background.

**Remarks:** ARRAYFILL is much better suited for filling of arrays.  
*addr:* see the {} function.

**See**

**Also:** MEMBFILL, MEMLFILL, ARRAYFILL

# MEMXOR Command

**Action:** performs a logical bit-wise XOR of two bit patterns in memory.

**Syntax:** MEMXOR src\_addr,dst\_addr,count  
*scr\_addr,dst\_addr: address*  
*count: iexp*

**Abbreviation:** -

**Explanation:** MEMXOR src\_addr, dst\_addr, count performs a logical XOR of two memory locations.

scr\_addr specifies the address of the source and dst\_addr the address of the destination location. count specifies the number of bytes to use at both locations. The logical XOR results in the target bits being set when the bits are set in either the source or target but not both.

**Remarks:**

```
SCREEN 3
DIM a!(15),b!(15)
a!(9)=-1,b!(9)=-1,b!(10)=-1,b!(12)=-1
SCREEN 3
@test
MEMXOR V:a!(0),V:b!(0),(DIM?(a!())+7)>>3
@test
REPEAT
UNTIL LEN(INKEY$)
//
PROCEDURE test
  FOR i%=0 TO 15
    PRINT STR$(a!(i%),2,0)'
  NEXT i%
  PRINT
  FOR i%=0 TO 15
```

```
PRINT STR$(b!(i%),2,0)'
NEXT i%
PRINT
RETURN

// Performs a bit-wise XOR of arrays
// a!() and b!(). The program prints
//
// 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 -1 -1 0 -1 0 0 0
// 0 0 0 0 0 0 0 0 0 0 -1 0 0 0 0 0 0
// 0 0 0 0 0 0 0 0 0 0 -1 0 -1 0 0 0 0

// This means that after the OR operation only
// the elements 10 and 12 in array b!()=TRUE.
```

**Remarks:**

This method of memory manipulation can be particularly handy for use with data bases. For example, if a data base contains variables which are used as flags to mark (-1) or not to mark (0) an attribute, this method of memory manipulation can be very helpful. Using MEMXOR an inquiry can be made to see if the markers apply to one or the other but not both attributes.

addr: see the {} function.

**See****Also:**

MEMOR, MEMAND

### MENU Command

**Action:** displays a menu bar.

**Syntax:** MENU m\$()  
*m\$(): the string array containing the menu entries*

**Abbreviation:** -

**Explanation:** If the graphic mode is on, MENU m\$() displays a menu bar on the screen. m\$() contains the menu title followed by menu entries and a null string to indicate the end of each menu. The following format must be observed when building a menu:

m\$(0):	name of the first menu
m\$(1):	first entry in the first menu
m\$(2):	second entry in the first menu
.	..
m\$(n):	last entry in the first menu
m\$(n+1)="":	marks the end of the first menu
m\$(n+2):	name of the second menu
m\$(n+3):	first entry in the second menu
.	..
m\$(n+m):	last entry in the second menu
m\$(n+m+1)="":	marks the end of the second menu

etc.

Do note, that MENU m\$() only displays the menu bar on the screen. The handling of menu entries must be performed by using the event function ON MENU GOSUB and the MENU() function.

A "hot key" for a menu entry can be specified by prefixing a letter with an underdash "\_". The pressing of this key will then return the index of the entry in the

menu array in MENU(0), the number of its menu title in MENU(7) and the number of the entry within this title in MENU(8).

**Example:**

```
DIM m$(20)
DATA Lissajous , Figure _1, Figure 2, Figure _3,
    Figure 4
DATA End ,"" , Names , Robert , Piere , Gustav
DATA Emile , Hugo ,!!
i%=-1
DO
    i%++
    READ m$(i%)           // read menu entries
LOOP UNTIL m$(i%)="!!"    // marks the end
m$(i%)=""                // menu termination
//
SCREEN 16                // turn on EGA mode
COLOR 8
PBOX 0,0,639,349
MENU m$()                // activate menu bar

//                        displays a menu bar
//                        with entries "Lissajous"
//                        and "Names".
```

**Remarks:**

The entries "Figure 1" and "Figure 3" can also be selected by using hotkeys.

**See****Also:**

MENU(), ON MENU GOSUB

### MENU() Function

**Action:** event management in pop-up menus and windows

**Syntax:** MENU (m)  
*m: iexp*

**Explanation:** The menu bar after the MENU m\$() command contains the pop-up menus with various menu entries. Such pop-up menus can be invoked in GFA-BASIC even outside of the menu bar by using the POP UP command. The MENU() function manages the pop-up menus and windows created with GFA-BASIC command OPENW. MENU(m) returns a value indicating which event has occurred. The values are assigned as follows:

MENU(1)=1	keyboard
MENU(4)	Information about the pressed key. The low byte contains the ASCII code and high byte the scan code of the pressed key
MENU(5)	status of the shift keys; Kbshift
MENU(1)=2	a mouse click outside of the current window (Top Window)
MENU(1)=7	returns the number of the clicked window
MENU(1)=3	a mouse click within the current window (Top Window)
MENU(1)=4	the close box of a window was activated
MENU(1)=5	the minimum size field in a window was activated
MENU(1)=6	the maximum size field in a window was activated



<b>MENU(1)=7</b>	the arrow up box in a window was activated
<b>MENU(1)=8</b>	the arrow down box in a window was activated
<b>MENU(1)=9</b>	the arrow left box in a window was activated
<b>MENU(1)=10</b>	the arrow right box in a window was activated
<b>MENU(1)=11</b>	the area above the vertical scroll bar was activated; Page up
<b>MENU(1)=12</b>	the area below the vertical scroll bar was activated; Page down
<b>MENU(1)=13</b>	the area to the left of the horizontal scroll bar was activated; Page left
<b>MENU(1)=14</b>	the area to the right of the scroll bar was activated; Page right
<b>MENU(1)=15</b>	the vertical scroll bar was moved
<b>MENU(7)</b>	position in the range from 0 to 1000
<b>MENU(1)=16</b>	the horizontal scroll bar was moved
<b>MENU(7)</b>	position in the range from 0 to 1000
<b>MENU(1)=17</b>	the title bar in a window was activated. If the window was moved,
<b>MENU(7)</b>	returns the new x-position
<b>MENU(8)</b>	returns the new y-position of the upper left corner of the window.
<b>MENU(1)=18</b>	the size box of a window was activated. If the size of the window was changed,
<b>MENU(7)</b>	returns the new width
<b>MENU(8)</b>	returns the new height of the window.
<b>MENU(1)=19</b>	the info line in a window was "clicked"
<b>MENU(1)=20</b>	a menu or a pop-up entry was selected.

**MENU(0)** returns the index of the menu entry in the entry field or the number of the entry in a pop-up menu.  
**MENU(7)** returns the number of the menu (1,2,...)  
**MENU(8)** returns the number of entry (1,2,...) in menu 1,2,...  
**MENU(1)=21** a rectangular segment of a window must be redrawn; Redraw Message  
**MENU(7)** returns the left x-coordinate of the window rectangle  
**MENU(8)** returns the upper y-coordinate of the window rectangle  
**MENU(9)** returns the width of the window rectangle  
**MENU(10)** returns the height of the window rectangle

The following always applies:

**MENU(2)** mouse x-position  
**MENU(3)** mouse y-position  
**MENU(4)** the status of the mouse keys:  
     **MENU(4)=0** no mouse key was pressed  
     **MENU(4)=1** the left mouse button was pressed  
     **MENU(4)=2** the right mouse button was pressed

### Example:

```

DIM m$(20)
DATA Lissajous , Figure 1 , Figure 2 , Figure 3 ,
      Figure 4
DATA End ,"" , Names , Robert , Piere , Gustav
DATA Emile , Hugo ,!!
i%=-1
DO
    i%++
    READ m$(i%)           // read menu entries
LOOP UNTIL m$(i%)="!!"    // marks the end
m$(i_%)=""               // menu termination
//
    
```

```
SCREEN 16           // turn on EGA mode
COLOR 8
PBOX 0,0,639,349
MENU m$( )         // activate menu bar
//
ON MENU GOSUB evaluation
DO
  ON MENU
LOOP
//
PROCEDURE evaluation
  LOCAL t_$
  t_$=TRIM$(a$(MENU(0)))
  IF t_$="End"
    MENU KILL
    SCREEN 3
    EDIT
  ELSE IF INSTR(t_$,"Figure")
    LOCATE 10,10
    PRINT " The entry ";t_$;" was selected"
  ENDIF
RETURN
```

```
// Creates a menu bar with entries "Lissajous" and
// "Names". When a menu is selected a branch is taken
// to PROCEDURE by using ON MENU GOSUB. The MENU(0)
// command determines which entry was selected. If
// "End" from the "Lissajous" menu was selected the
// menu is removed using MENU KILL, the program
// returns to the TEXT mode and the GFA-BASIC
// interpreter is invoked.
// If an entry containing the "Figure" string was
// selected from the "Lissajous" menu, the selected
// entry is written out. Otherwise, no further events
// are evaluated..
```

## Commands and functions

---

**Remarks:**

-

**See**

**Also:**

MENU, ON MENU GOSUB, GETEVENT,  
PEEKEVENT

## MENU KILL Command

**Action:** removes a pop-up menu

**Syntax:** MENU KILL

**Abbreviation:** -

**Explanation:** MENU KILL removes the current pop-up menu without deleting it. The ON MENU GOSUB is also deactivated.

**Example:**

```
DIM m$(20)
DATA Lissajous , Figure 1 , Figure 2 , Figure 3 ,
      Figure 4
DATA End ,"" , Names , Robert , Piere , Gustav
DATA Emile , Hugo ,!!
i%=-1
DO
  i%++
  READ m$(i%)           // read menu entries
LOOP UNTIL m$(i%)="!!"  // marks the end
m$(i%)=""               // menu termination
//
SCREEN 16               // turn on EGA mode
COLOR 8
PBOX 0,0,639,349
MENU m$()              // activate menu bar
//
ON MENU GOSUB evaluation
DO
  ON MENU
LOOP UNTIL MOUSEK AND 2
//
PROCEDURE evaluation
  LOCAL t_$
  t_$=TRIM$(m$(MENU(0)))
```

## Commands and functions

---

```
IF t$="ENDE"  
  MENU KILL  
  SCREEN 3  
  EDIT  
ENDIF  
RETURN
```

```
// Displays a menu bar with entries "Lissajous" and  
// "Names".  
// When the "End" entry in the "Lissajous" menu is  
// selected the menu is closed and the GFA-BASIC  
// editor is invoked.
```

**Remarks:** -

**See**

**Also:**

MENU(), ON MENU GOSUB

## MFREE() Function

**Action:** releases reserved memory.

**Syntax:** MFREE(addr)  
*addr: address*

**Abbreviation:** -

**Explanation:** MFREE(addr) releases memory previously reserved with MALLOC(n). addr contains the address of memory which was allocated with MALLOC(). In case of error MFREE() returns a negative number.

**Example:**

```
PRINT FRE(0)
a%=MALLOC(2000)
PRINT FRE(0)
MFREE(a%)
PRINT FRE(0)
```

**Remarks:** addr: see the {} function.

**See**

**Also:** MALLOC(), MSHRINK()

## MID\$ Command

**Action:** moves a string expression of specified length to the chosen place in a character string.

**Syntax:** MID\$(a\$,p%,l%)=b\$  
*a\$:* *svar*  
*b\$:* *sexp*  
*p%,l%:* *iexp*

**Abbreviation:** -

**Explanation:** MID\$(a\$,p%,l%)=b\$ moves l% characters from b\$, to position p% (in a\$) to a\$. If l% is left out, again, as many characters as possible are moved from b\$ to a\$. The length and address of a\$ are not changed.

**Example:**

```
a$=STRING$(15,"-")
b$="Hello GFA"
PRINT a$'LEN(a$) // prints ----- 15

PRINT b$'LEN(b$) // prints Hello GFA      9

MID$(a$,3)=b$
PRINT a$'LEN(a$) // prints Hello GFA      15

MID$(a$,9,5)=b$
PRINT a$'LEN(a$) // prints o GFA          15
```

**Remarks:** -

**See**

**Also:** LSET, RSET



## MID\$() String function

**Action:** starting from position *p*, returns the next *m* characters of a string expression.

**Syntax:** MID\$(*a\$,p%,m%*)  
*a\$*: *sexp*  
*m%,p%*: *iexp*

**Abbreviation:** -

**Explanation:** MID\$(*a\$,p%,m%*) returns, starting from position *p%* (inclusive), up to *m%* characters of the string expression *a\$*. If *m%* is not given, the whole string from position *p%* is returned.

**Example:**

PRINT MID\$("Hello GFA",7,5)	//	prints	GFA
PRINT MID\$("Hello GFA",1,5)	//	prints	Hello
PRINT MID\$("Hello GFA",3)	//	prints	llo GFA

**Remarks:** -

**See**

**Also:** LEFT\$(), RIGHT\$()

### MIN() Function

**Action:** returns the smallest argument.

**Syntax:** MIN(x1,x2[,x3,...,xn]) or

MIN(x1\$,x2\$[,x3\$,...,xn\$])

*x1,x2,...: aexp*

*x1\$,x2\$,...: sexp*

**Explanation:** MIN(x1,x2[,x3,...,xn]) determines the smallest of numeric expressions x1,..xn. In case of strings each single ASCII code in a string is compared character by character.

**Example:**

```
PRINT MIN(2,PI,SQR(64)) // prints 2
PRINT MIN("ABBBBBB","B") // prints ABBBBBB
```

**Remarks:** -

**See**

**Also:** MAX()

## MIRROR\$() String function

**Action:** generates a string which is a mirror image of the given character expression.

**Syntax:** MIRROR\$(a\$)  
*a\$:* *sexp*

**Abbreviation:** -

**Explanation:** MIRROR\$ reverses the characters in a string.

**Example:** PRINT MIRROR\$("Hello GFA")// prints AFG olleH

**Remarks:** -

**See**

**Also:** XLATE\$()

### MKD\$() Function

**Action:** converts a 64-bit floating point expression to eight characters in IEEE double format.

**Syntax:** MKD\$(x)  
*x:* *aexp*

**Explanation:** Creates an eight characters long string from a number internally stored in IEEE double format.

**Example:**

```
PRINT MKD$(1001.1001)

OPEN "R",#1,"Test",19
FIELD #1,1 AS a$,2 AS b$,4 AS c$, 4 AS d$,8 AS e$
a$=CHR$(123)
b$=MKI$((1234)
c$=MKL$(12345678)
d$=MKS$(1.23)
e$=MKD$(1.23)
PUT #1,1
//
GET #1,1
PRINT ASC(a$)'CVI(b$)'CVL(c$)'CVS(d$)'CVD(e$)

// Prints
// 123 1234 12345678 123000019.. 123
```

**Remarks:** The order of the bytes depends on the processor. For 80x86 or 8088 processors LSB (least significant byte) is converted first and MSB (most significant byte) is converted last.

MKD\$() is the reverse function of CVD().

**See**

**Also:** ASC(), CVI(), CVL(), CVS(), CVD(), CHR\$(), MKI\$(), MKL\$(), MKS\$()

## MKDIR Command

<b>Action:</b>	creates a directory.
<b>Syntax:</b>	MKDIR a\$ <i>a\$: sexp; directory name</i>
<b>Abbreviation:</b>	mkdi
<b>Explanation:</b>	MKDIR a\$ (make directory) creates a directory with name a\$.
<b>Example:</b>	MKDIR "C:\TEST"  // Creates a directory called TEST on drive C
<b>Remarks:</b>	-
<b>See Also:</b>	RMDIR

### MKI\$() Function

**Action:** converts a 16-bit integer expression to two characters.

**Syntax:** MKI\$(m)  
*m: iexp*

**Explanation:** Creates a two character long string from an integer.

**Example:**

```
PRINT MKI$(1000)
PRINT MKI$(CVI("Hello GFA"))

OPEN "R",#1,"Test",19
FIELD #1,1 AS a$,2 AS b$,4 AS c$, 4 AS d$,8 AS e$
a$=CHR$(123)
b$=MKI$((1234)
c$=MKL$(12345678)
d$=MKS$(1.23)
e$=MKD$(1.23)
PUT #1,1
//
GET #1,1
PRINT ASC(a$)'CVI(b$)'CVL(c$)'CVS(d$)'CVD(e$)

// Prints
// 123 1234 12345678 123000019.. 123
```

**Remarks:** For more information refer to the MKS\$() function.  
MKI\$() is the reverse function of CVI().

**See Also:** ASC(), CVI(), CVL(), CVS(), CVD(), CHR\$(),  
MKL\$(), MKS\$(), MKD\$()

## MKS\$() Function

**Action:** converts a 32-bit floating point expression into IEEE single format in four characters.

**Syntax:** MKS\$(x)  
*x:* *aexp*

**Explanation:** Creates a four character long string from a number internally stored in IEEE single format.

**Example:**

```
PRINT MKS$(100.1)

OPEN "R",#1,"Test",19
FIELD #1,1 AS a$,2 AS b$,4 AS c$, 4 AS d$,8 AS e$
a$=CHR$(123)
b$=MKI$((1234)
c$=MKL$(12345678)
d$=MKS$(1.23)
e$=MKD$(1.23)
PUT #1,1
//
GET #1,1
PRINT ASC(a$)'CVI(b$)'CVL(c$)'CVS(d$)'CVD(e$)

// Prints
// 123 1234 12345678 123000019.. 123
```

**Remarks:** The order of the bytes depends on the processor. For 80x86 or 8088 processors LSB (least significant byte) is converted first and MSB (most significant byte) is converted last.

MKS() is the reverse function of CVS().

**See**

**Also:** ASC(), CVI(), CVL(), CVS(), CVD(), CHR\$(), MKI\$(), MKL\$(), MKD\$()

### MOD Function

**Action:** calculates the modulus of an integer expression based on the second integer expression.

**Syntax:** `i MOD j`  
`i,j: iexp`

**Explanation:** `i MOD j` calculates the modulus of the integer expression `i` based on the integer expression `j` and, optionally, writes this value to a variable.

**Example:**

```
PRINT 42 MOD 6      // prints    0
1%=42 MOD 5
PRINT 1%            // prints    2
```

**Remarks:** -

**See**

**Also:** `ADD()`, `SUB()`, `DIV()`, `MOD()`, `PRED()`, `SUCC()`



## MOD() Function

**Action:** calculates the modulus of an integer expression based on the second integer expression.

**Syntax:** MOD(*i,j*)  
*i,j: iexp*

**Explanation:** MOD(*i,j*) calculates the modulus of the integer expression *i* based on the integer expression *j* and, optionally, writes this value to a variable.

**Example:**

```
PRINT MOD(42,6)           // prints    0
1%=MOD(42,5)
PRINT 1%                  // prints    2
```

**Remarks:** -

**See**

**Also:** ADD(), SUB(), DIV(), MOD(), PRED(), SUCC()

## MODE Command

**Action:** sets the display mode of date and formatted numerical output.

**Syntax:** MODE n

**Abbreviation:** -

**Explanation:** The MODE command allows for two types of formatted numerical output: decimal point and comma as a divider for thousands (English default) or decimal comma and point as a divider for thousands (continental mode). These modes apply to PRINT USING, PRINT AT...USING and STR\$(). For the date display the choice is between a period and a backslash. These modes apply to DATE\$, SETTIME, DATE\$= and FILES.

The parameter n can have the values between 0 and 3. The corresponding displays are as follows:

Parameter n	USING	DATE\$
MODE 0	#,###.##	24.07.1990
MODE 1	#,###.##	07/24/1990
MODE 2	#.###,##	24.07.1990
MODE 3	#.###,##	07/24/1990

**Example:** -

**Remarks:** -

**See  
Also:** -

## Monitor Command

**Action:** invokes "foreign" subroutines.

**Syntax:** MONITOR[n]  
*n: iexp*

**Abbreviation:** mon

**Explanation:** MONITOR [n] calls interrupt \$3 and passes the value n in processor registers AX and DX. The command is intended for inserting of breakpoints in compiled programs.

**Example:** -

**Remarks:** -

**See  
Also:** -

## MOVEW # Command

**Action:** moves a window.

**Syntax:** MOVEW #n,x,y  
*n,x,y: iexp*

**Abbreviation:** -

**Explanation:** MOVEW #n,x,y moves the window specified in n (1, 2, 3 or 4) so that its upper left corner is at coordinates x,y.

**Example:**

```
SCREEN 16 // EGA mode
OPENW #1,10,10,200,100,-1
OPENW #2,15,15,200,100,-1
REPEAT
UNTIL LEN(INKEY$)
MOVEW #2,50,50
REPEAT
UNTIL LEN(INKEY$)
CLOSEW #1
CLOSEW #2
EDIT

// Draws two windows on the screen and waits for a
// keypress. The second window is then moved.
```

**Remarks:** Only the top window can be moved.

**See**

**Also:** OPENW, CLOSEW, TITLEW, INFOW, SIZEW,  
FULLW, CLEARW, TOPW

## MSHRINK() Function

**Action:** decrements reserved memory

**Syntax:** MSHRINK(addr,n)  
*addr: address*  
*n: iexp*

**Explanation:** MSHRINK(addr,n) shrinks memory previously reserved with MALLOC(n) and releases it. addr contains the address of memory allocated with MALLOC() and n contains the size in bytes. If the function is invoked correctly, MSHRINK() returns an address. In case of an error it returns a negative value.

**Example:**

```
PRINT FRE(0)
a%=MALLOC(5000)
PRINT FRE(0)
MSHRINK(a%,2000)
PRINT FRE(0)
```

**Remarks:** addr: see the {} function.

**See**

**Also:** MALLOC(), MFREE()

## MUL Command

**Action:** multiplies a numeric variable with a numeric expression.

**Syntax:** MUL *x*,*y*  
*x*: *avar*  
*y*: *aexp*

**Abbreviation:** -

**Explanation:** MUL *x*,*y* multiplies the value in variable *x* with the expression *y*.

**Example:**

```
x=6
MUL x,9
PRINT x           // prints 54
```

**Remarks:** Although MUL can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of MUL *x*,*y*

```
x = x*y
x := x*y    or
x *= y
```

can be used also.

When integer variables are used MUL doesn't test for overflow!

**See**

**Also:** DEC, INC, ADD, SUB, DIV, ++, --, +=, -=, \*=, /=

## MUL() Function

**Action:** multiplies two or more integer expressions.

**Syntax:** MUL(*i,j[,m,...]*)  
*i,j,m,...: iexp*

**Explanation:** MUL(*i,j[,m,...]*) calculates the product of integer expressions *i*, *j*, *m*,... and, optionally, stores it in a variable.

**Example:**

```
PRINT MUL(6,9)           // prints 54
PRINT MUL(2,3,ADD(3,4))  // prints 42
1%=MUL(6,9)
PRINT 1%                 // prints 54
```

**Remarks:** The ADD(), SUB(), MUL() and DIV() functions can be mixed freely with each other. For example:

```
1%=ADD(5^3,4*20-3)      or
1%=ADD(5^3,SUB(MUL(4,20),3))
```

**See**

**Also:** ADD(), SUB(), DIV(), MOD(), PRED(), SUCC()

### NAME...AS Command

**Action:** renames a file.

**Syntax:** NAME old\$ AS new\$  
*old\$,new\$: sexp; old and new file names*

**Abbreviation:** nam old\$ as new\$

**Explanation:** -

**Example:**

```
old$="A:\TEST.TXT"
new$="A:\NEW.TXT"
IF EXIST old$
    NAME old$ AS new$
ENDIF

// Checks if the file with the name TEST.TXT exists
// on drive A and renames it to NEW.TXT.
```

**Remarks:** RENAME...AS is synonymous with NAME...AS and can be used instead.

**See**

**Also:** RENAME...AS



## NEW Command

<b>Action:</b>	deletes the current program in main memory.
<b>Syntax:</b>	NEW
<b>Abbreviation:</b>	-
<b>Explanation:</b>	NEW can only be used from direct mode or by selecting the corresponding GFA-BASIC editor function and it will then delete the program in memory.
<b>Example:</b>	-
<b>Remarks:</b>	-
<b>See Also:</b>	-





### OCT\$() Function

**Action:** converts an integer expression to octal representation.

**Syntax:** OCT\$(m[,n])  
*m,n: iexp*

**Explanation:** After conversion the octal representation of integer expression *m* is returned as a plain string. The parameter *n* is optional and determines how many places should be used. If *n* is greater than the number of places needed to represent *m* the converted number is padded with leading zeros.

**Example:**

```
PRINT OCT$(17)           // prints    21
PRINT OCT$(25,6)         // prints    000031
```

**Remarks:** -

**See**

**Also:** BIN\$(), HEX\$(), DEC\$()

## ODD() Numeric function

**Action:** tests if a numeric expression is odd and returns -1 (true) if it is, or 0 if the expression is even.

**Syntax:** ODD(*x*)  
*x*: *aexp*

**Explanation:** -

**Example:**

```
x=2
PRINT ODD(x*3)           // prints    0
PRINT ODD(x*3+1)        // prints   -1
```

**Remarks:** -

**See**

**Also:** EVEN()

## ON BREAK Command

**Action:** turns on the operating system monitoring of Control + Break key combination.

**Syntax:** ON BREAK

**Abbreviation:** -

**Explanation:** ON BREAK should only be used together with ON BREAK GOSUB (refer to the command itself). The ON BREAK command turns back on the normal operating system reaction (program break).

**Example:**

```
ON BREAK GOSUB break_reac
SCREEN 3
PRINT "Press Control+Break now"
DO
LOOP
//
PROCEDURE break_reac
    PRINT "O.K."
    ON BREAK
RETURN

// You are requested to press the Control+Break key
// combination. When that happens the program jumps
// to PROCEDURE proc_reac, which prints the O.K. mes-
// sage and turns the normal BREAK routine back on.
```

**Remarks:** -

**See**

**Also:** ON BREAK GOSUB, ON BREAK CONT, ON ERROR, ON ERROR GOSUB

## ON BREAK CONT Command

**Action:** turns off the Control + Break key combination.

**Syntax:** ON BREAK CONT

**Abbreviation:** -

**Explanation:** ON BREAK CONT turns off the Control + Break key combination, which means that the pressing of this key combination has no effect.

**Example:**

```
ON BREAK CONT
SCREEN 3
PRINT "Press Control+Break now"
DO
LOOP
```

```
// You are requested to press the Control+Break key
// combination. When that happens the program "hangs
// up" in an endless loop which cannot be broken by
// pressing these two keys anymore.
```

**Remarks:** -

**See**

**Also:** ON BREAK, ON BREAK GOSUB, ON ERROR, ON ERROR GOSUB

## ON BREAK GOSUB Command

**Action:** performs a branch when Control+Break key combination is pressed.

**Syntax:** ON BREAK GOSUB proc  
*proc: name of a PROCEDURE to jump to*

**Abbreviation:** -

**Explanation:** When Control+Break is pressed ON BREAK GOSUB proc jumps to the PROCEDURE specified in proc.

**Example:**

```
ON BREAK GOSUB break_reac
SCREEN 3
PRINT "Press Control+Break key combination now"
DO
LOOP
//
PROCEDURE break_reac
    PRINT "O.K."
    ON BREAK
RETURN

// You are requested to press the Control+Break key
// combination. When that happens the program jumps
// to the PROCEDURE proc_reac, which prints the O.K.
// message and turns the normal BREAK routine back
// on.
```

**Remarks:** -

**See**

**Also:** ON BREAK, ON BREAK CONT, ON ERROR, ON ERROR GOSUB



## ON ERROR Command

**Action:** turns on the reporting of error messages by the operating system or GFA-BASIC.

**Syntax:** ON ERROR

**Abbreviation:** -

**Explanation:** ON ERROR is used together with ON ERROR GOSUB (refer to the command itself). The ON ERROR command turns back on the normal operating system or GFA-BASIC reaction to an error (error messages and program break) which was redirected with ON ERROR GOSUB.

**Example:**

```
SCREEN 3
n%=4
DIM a(n%,n%),b(n%,n%)
DATA 1,2,3,4
DATA 2,4,6,8
DATA 3,6,9,12
DATA 4,8,12,16
MAT READ b()
MAT DET x=b()
PRINT x
PRINT
ON ERROR GOSUB inv_error
MAT INV a(,)=b()
IF s!
    MAT CLEAR a()
ELSE
    MAT PRINT a(),13,5
ENDIF
EDIT
//
```

## Commands and functions

---

```
PROCEDURE inv_error
  PRINT "The matrix is singular"
  s!=TRUE
  ON ERROR
RETURN

// The matrix b() is initialized with values from the
// DATA lines and the determinant of b() is cal-
// culated and printed out. Since this is 0, the
// inverse of b() is not possible. The command MAT
// INV a()=b() would therefore report 'Division by
// zero' error and interrupt the program. By using ON
// ERROR GOSUB inv_error this error is intercepted
// and the corresponding PROCEDURE is invoked. It
// reports the error message 'The matrix is singular'
// and sets a flag which can be interrogated by the
// program. Finally, the ON ERROR turns the normal
// error routine back on.
```

**Remarks:**

-

**See**

**Also:**

ON BREAK, ON BREAK GOSUB, ON BREAK  
CONT, ON ERROR GOSUB, RESUME

## ON ERROR GOSUB Command

**Action:** jumps to a routine when an error occurs

**Syntax:** ON ERROR GOSUB proc  
*proc: name of a PROCEDURE to jump to  
in case of an error*

**Abbreviation:** -

**Explanation:** When an error occurs (for example: division by zero or square root of a negative number) ON ERROR GOSUB proc jumps to the PROCEDURE specified in proc.

**Example:**

```
SCREEN 3
n%=4
DIM a(n%,n%),b(n%,n%)
DATA 1,2,3,4
DATA 2,4,6,8
DATA 3,6,9,12
DATA 4,8,12,16
MAT READ b()
MAT DET x=b()
PRINT x
PRINT
ON ERROR GOSUB inv_error
MAT INV a()=b()
IF s!
    MAT CLEAR a()
ELSE
    MAT PRINT a(),13,5
ENDIF
EDIT
//
```

## Commands and functions

---

```
PROCEDURE inv_error
  PRINT "The matrix is singular"
  s!=TRUE
  ON ERROR
RETURN
```

```
// The matrix b() is initialized with values from the
// DATA lines and the determinant of b() is calculated and printed out. Since this is 0, the
// inverse of b() is not possible. The command MAT
// INV a()=b() would therefore report 'Division by
// zero' error and interrupt the program. By using ON
// ERROR GOSUB inv_error this error is intercepted
// and the corresponding PROCEDURE is invoked. It
// reports the error message 'The matrix is singular'
// and sets a flag which can be interrogated by the
// program. Finally, the ON ERROR turns the normal
// error routine back on.
```

**See**

**Also:**

ON BREAK, ON BREAK GOSUB, ON BREAK  
CONT, ON ERROR RESUME

## ON...GOSUB Conditional directive

**Action:** performs a branch to a subroutine specified after GOSUB, conditional on the value of the expression after ON.

**Syntax:** ON n GOSUB proc1,proc2,...,procm  
*n:* *iexp*  
*proc1,...,procm:* *procedurenames (without parameters!)*

**Abbreviation:** on n gos ...

**Explanation:** A branch to the n-th procedure is performed, depending on the value of n. If the value of n is less than 1 or greater than the number of procedures specified after GOSUB, no subroutine will be invoked. If n is not an integer, a TRUNC(n) will be performed first and, if needed, a branch will then be taken. After a subroutine invoked using ON...GOSUB is executed the program continues with the first statement after ON...GOSUB. If a subroutine is not invoked, the execution immediately continues with the first statement after ON...GOSUB.

No parameters can be passed to a subroutine invoked with ON...GOSUB.

**Example:** n%=3  
ON n% GOSUB p1,p2,p3,p4,p5,p6  
n%=2  
ON n% GOSUB p1,p2,p3,p4,p5,p6

// The subroutine p3 is called first, followed by the  
// subroutine p6.

**Remarks:** The program flow defined using ON...GOSUB can also be achieved with SELECT...ENDSELECT or IF...ENDIF. The difference is that with ON...GOSUB a subroutine is always performed, while CASE, IF, ELSE IF etc. can be followed by a simple block of statements.

**See**

**Also:** IF...ENDIF, SELECT...ENDSELECT

## ON MENU Command

**Action:** monitors events in menus and windows

**Syntax:** ON MENU

**Explanation:** ON MENU monitors the occurrence of events in the menu bar, pop-up menus and windows. Before this command it should be established which event should go to which PROCEDURE. The following commands are available to do this:

```
ON MENU GOSUB
ON MENU KEY GOSUB and
ON MENU MESSAGE GOSUB
```

**Example:**

```
DIM m$(20)
DATA Lissajous , Figure 1 , Figure 2 , Figure 3 ,
Figure 4
DATA End ,"" , Names , Robert , Piere , Gustav
DATA Emile , Hugo ,!!
i%=-1
DO
  i%++
  READ m$(i%)           // read menu entries
LOOP UNTIL m$(i%)="!!"  // marks the end
m$(i%)=""               // menu termination
//
SCREEN 16               // turn on EGA mode
COLOR 8
PBOX 0,0,639,349
MENU m$()              // activate menu bar
//
ON MENU GOSUB eval_pop_up
ON MENU KEY GOSUB eval_key
ON MENU MESSAGE GOSUB eval_window
DO
```

## Commands and functions

---

```
ON MENU
LOOP
//
PROCEDURE eval_pop_up
    // evaluate events in pop-up menus
RETURN
//
PROCEDURE eval_key
    // evaluate events from the keyboard
RETRUN
//
PROCEDURE eval_window
    // evaluates the events in the windows
RETURN

// Displays a menu bar with entries "Lissajous" and
// "Names". When a menu entry is selected the
// PROCEDURE eval_pop_up is executed, when keyboard
// input is detected the PROCEDURE eval_key is
// invoked and when an event in the window occurs the
// PROCEDURE eval_window is performed.
```

**Remarks:**

-

**See**

**Also:**

MENU, MENU() ON MENU GOSUB, ON MENU  
KEY GOSUB, ON MENU MESSAGE GOSUB



## ON MENU GOSUB Command

**Action:** a branch based on a menu event

**Syntax:** ON MENU GOSUB *proc*  
*proc*: subroutine

**Explanation:** ON MENU GOSUB evaluate, branches to the PROCEDURE evaluate, when a menu entry is selected from the menu bar created with MENU m\$(). The actual event is determined by the command ON MENU. The function MENU() can then be used within the PROCEDURE evaluate to determine which entry was selected.

Do note, that ON MENU GOSUB will only branch to the specified subroutine and perform no other function.

**Example:**

```
DIM m$(20)
DATA Lissajous , Figure 1 , Figure 2 , Figure 3 ,
Figure 4
DATA End ,"" , Names , Robert , Piere , Gustav
DATA Emile , Hugo ,!!
i%=-1
DO
  i%++
  READ m$(i%)           // read menu entries
LOOP UNTIL m$(i%)="!!"  // marks the end
m$(i%)=""               // menu termination
//
SCREEN 16               // turn on EGA mode
COLOR 8
PBOX 0,0,639,349
MENU m$()              // activate menu bar
//
ON MENU GOSUB evaluate
```

## Commands and functions

---

```
DO
  ON MENU
LOOP
//
PROCEDURE evaluate
  // menu evaluation
RETURN

// Displays a menu bar with the entries "Lissajous"
// and "Names". When a menu entry is selected a
// branch is taken to the PROCEDURE evaluate.
```

### Remarks:

-

### See

### Also:

MENU, MENU()

## ON MENU KEY GOSUB Command

**Action:** a branch based on a menu event

**Syntax:** ON MENU KEY GOSUB *proc*  
*proc: subroutine*

**Explanation:** ON MENU KEY GOSUB *eval\_key* monitors the keyboard and branches to the PROCEDURE *eval\_key* if a key was pressed during the ON MENU command.

Do note, that ON MENU KEY GOSUB will only branch to the specified subroutine and perform no other function.

**Example:** ON MENU KEY GOSUB *eval\_key*

```
DO
```

```
ON MENU
```

```
LOOP UNTIL MOUSEK AND 2
```

```
//
```

```
PROCEDURE eval_key
```

```
PRINT "Special keys : "; MENU(13)
```

```
PRINT "ASCII code : ";BYTE(MENU(14))
```

```
PRINT "SCAN code : ";SHR(MENU(14),8)
```

```
PRINT
```

```
RETURN
```

```
// Monitors the keyboard and, when a key is pressed,  
// returns the current status of the special keys as  
// well as the ASCII and scan codes of the pressed  
// key.
```

**Remarks:** -

**See**

**Also:** MENU, MENU(), ON MENU, ON MENU GOSUB,  
ON MENU MESSAGE GOSUB

## ON MENU MESSAGE GOSUB Command

**Action:** a branch based on a menu event

**Syntax:** ON MENU MESSAGE GOSUB *proc*  
*proc*: subroutine

**Explanation:** ON MENU MESSAGE GOSUB evaluate, monitors the message buffer and branches to the PROCEDURE *eval\_message*, when there is a message in the message buffer. The monitoring of the message buffer happens with all ON MENU commands.

Do note, that ON MENU MESSAGE GOSUB will only branch to the specified subroutine and perform no other function.

**Example:**

```
SCREEN 16
OPENW #1,10,10,400,300,-1
//
ON MENU MESSAGE GOSUB eval_message
DO
    ON MENU
LOOP UNTIL MOUSEK AND 2
//
PROCEDURE eval_message
    LOCATE 10,10
    SELECT MENU(1)
    CASE 4
        PRINT "the window will be closed"
    CASE 16
        PRINT " the bottom scroll bar will be moved"
    CASE 17
        PRINT " the top scroll bar will be moved"
```

```
DEFAULT  
  PRINT CHR$(7)  
ENDSELECT  
RETURN
```

```
// Monitors the message buffer and prints a notice ,  
// if the close box in a window was activated or the  
// bottom or top scroll bar was moved.  
// Otherwise the bell is sounded.
```

**Remarks:**

-

**See**

**Also:**

MENU, MENU(), ON MENU, ON MENU GOSUB,  
ON MENU KEY GOSUB, PEEKEVENT,  
GETEVENT

# OPEN Command

**Action:** opens a data channel to a file or a peripheral device.

**Syntax:** OPEN mode\$,#n,name\$[count]  
mode\$,name\$: sexp  
n,count: iexp

**Abbreviation:** o mode\$,#n,name\$[count]

**Explanation:** OPEN mode\$,#n,name\$[count] opens the channel n in mode mode\$ to the file name\$. n can assume values between 0 and 99. This channel number must always be specified when the file is being accessed. The various access modes are as follows:

mode\$	effect
"o"	(output) opens a file for writing. If the file exists its data is deleted and if it doesn't a new file is created.
"u"	(update) opens an existing file for both reading and writing.
"i"	(input) opens a file for reading.
"a"	(append) enables adding of data to an existing file. It moves the data pointer to the end of file.
"r"	(random access) opens a file for arbitrary reading and writing. The description of this type of file can be found in the FIELD command.
name\$	contains the pathname of the file. Instead of a file name a peripheral id can also be specified.

The meaning of which is as follows:

<b>LPT1:,...LPT4:</b>	parallel port (Centronics)
<b>COM1:,...COM4</b>	serial port (RS232)
<b>CON:</b>	keyboard/screen

count is used only with random access files and it contains the length of a record.

**Example:**

```
OPEN "o",#1,"A:\TEST" // opens the file TEST
FOR i%=1 TO 5          // on drive A and
  PRINT #1, STR$(i%)   // writes numbers from 1
NEXT i%                // to 5 to it.
CLOSE #1

OPEN "a",#2,"A:\TEST" // opens the existing file
//                     TEST
FOR i%=20 TO 30        // on drive A
  PRINT #1, STR$(i%)   // and appends the numbers
NEXT i%                // from 20 to 30 to it.
CLOSE #2               //

OPEN "o",#1,"CON:"     // opens the channel #1
FOR i%=1 TO 20         // and writes the numbers
  PRINT #1, i%         // from 1 to 20
NEXT i%                // to screen
CLOSE #1
```

**Remarks:**

-

**See**

**Also:**

CLOSE

### OPENW# Command

**Action:** opens a window.

**Syntax:** OPENW #*n,x,y,w,h,attr*  
*n,x,y,w,h,attr: iexp*

**Abbreviation:** -

**Explanation:** When in a graphic mode, OPENW #*n,x,y,w,h,attr* opens the window with number *n*, where *n* can assume the values 0, 1, 2, 3 and 4. The upper left corner of the window is anchored at the coordinates specified with *x* and *y*. The window has the width *w* and the height *h*. By using *attr* the following window attributes can be specified:

**attr:** word sized bitmap. The set bits are interpreted as follows:

Bit	Value	Meaning
0	1	vertical slider
1	2	arrow up, arrow down
2	4	horizontal slider
3	8	arrow left, arrow right
4	16	title line
5	32	close box
6	64	minimise box
7	128	maximise box
8	256	info line
9	512	size box

**attr = -1** draws all areas

**attr = 0** draws one rectangle only (3D)

OPENW #0 does not actually open a window, but only moves the coordinate reference point to prevent the menu bar from being overwritten. Following an



OPENW #0 the upper 19 pixel rows are no longer available for graphic and text output.

**Example:**

```
SCREEN 16           // EGA mode
OPENW #1,10,10,200,100,-1
REPEAT
  a$=inkey$
UNTIL a$=chr$(27)
CLOSEW #1
SCREEN 3
EDIT
```

// Draws a window on the screen.

**Remarks:**

The management of events within the window (for example, close box or the scroll bars) must be performed by using ON MENU MESSAGE GOSUB, ON MENU and MENU().

**See**

**Also:**

CLOSEW, TITLEW, INFOW, SIZEW, TOPW, FULLW, CLEARW

### OPTION BASE Command

**Action:** sets the starting offset for row/column indexing of arrays.

**Syntax:** OPTION BASE 0 or  
OPTION BASE 1

**Abbreviation:** -

**Explanation:** The OPTION BASE command sets the starting offset for row/column indexing of arrays. In case of OPTION BASE 0 the indexing starts with element 0, and in case of OPTION BASE 1 with element 1. OPTION BASE 0 is the default.

**Example:**

```
SCREEN 3
DIM a%(10)
FOR i%=10 DOWNT0 0
    a%(i%)=i%
NEXT i%
PRINT "COUNT-DOWN :"'
FOR i%=10 DOWNT0 0
    PRINT a%(i%)'
NEXT i%
PRINT
//
OPTION BASE 1
PRINT "COUNT-DOWN :"'
FOR i%=10 DOWNT0 0
    PRINT a%(i%)'
NEXT i%
```

```
// Writes COUNT-DOWN and then the digits from 10 to 0
// on the screen. Following that COUNT-DOWN is
// written again and then the numbers from 10 to 1.
// The error message "Invalid array index" appears
```

```
// next since, following OPTION BASE 1, array a%()  
// has no element a%(0) anymore.
```

**Remarks:** -

**See**

**Also:** MAT BASE

### OR Function

**Action:** performs a logical bit-wise OR on two bit patterns.

**Syntax:** `i OR j`  
`i,j: iexp`

**Explanation:** `i OR j` sets only the bits which are set in at least one of the two operands `i` or `j`.

**Example:**

```
PRINT BIN$(3,4) // prints 0011
PRINT BIN$(10,4) // prints 1010
PRINT BIN$(3 OR 10,4) // prints 1011
```

**Remarks:** `|` is synonymous with `OR` and can be used instead:

```
PRINT BIN$(3 | 10,4) // prints 1011
```

**See**

**Also:** `AND`, `XOR`, `IMP`, `EQV`, `&`, `|`, `~`

## OR() Function

**Action:** performs a logical bit-wise OR on two bit patterns.

**Syntax:** OR(*i,j*)  
*i,j*: *iexp*

**Explanation:** OR(*i,j*) sets only the bits which are set in at least one of the two operands *i* or *j*.

**Example:**

PRINT BIN\$(3,4)	// prints	0011
PRINT BIN\$(10,4)	// prints	1010
PRINT BIN\$(OR(3,10),4)	// prints	1011

**Remarks:** -

**See**

**Also:** AND(), XOR(), IMP(), EQV()

## OUT #n Command

**Action:** sends a byte to an already opened file.

**Syntax:** OUT #n,a[,b,c...]  
*n:* *iexp; channel number*  
*a,b,c...:* *iexp*

**Abbreviation:** -

**Explanation:** OUT #n writes a byte to a previously opened file. The numerical expression n contains the channel number (from 0 to 99) used to access the file.

**Example:**

```
OPEN "o",#1,"A:\TEST.DAT"  
FOR i%=1 to 20  
    OUT(#1),128  
NEXT i%  
CLOSE #1
```

```
// Opens the file TEST.DAT on drive A and writes the  
// value 128 to it 20 times from inside a FOR...NEXT  
// loop.
```

**Remarks:** OUT | #n is synonymous with OUT #n and can be used instead.

OUT & #n can be used to write a word (two bytes),  
OUT% #n to write a longword (four bytes) to  
successive port addresses.

**See**

**Also:** INP

## OUT PORT Command

<b>Action:</b>	hardware access
<b>Syntax:</b>	<code>OUT PORT n,m</code> <i>n: iexp; port number</i> <i>m: iexp</i>
<b>Abbreviation:</b>	-
<b>Explanation:</b>	OUT PORT n,m writes a byte to a hardware port register, RTC for example.
<b>Example:</b>	This command implies an intimate knowledge of the hardware and is not portable.
<b>Remarks:</b>	OUT ^ n,m or OUT  PORT n,m or OUT  ^ n,m are synonymous with OUT PORT n,m and can be used instead. OUT& ^ n can be used to write a word (two bytes), OUT% PORT n,m to write a longword (four bytes) to successive port addresses.
<b>See Also:</b>	INP(PORT)

### Parameter list

**Action:** the part of a PROCEDURE, FUNCTION or DEFFN where variables and/or arrays (parameters) are declared and passed during a call.

**Syntax:** PROCEDURE name(v1[,...,vn][ VAR arr1(),...,arrn(),  
r1[,...,rn])

or

FUNCTION name(v1[,...,vn][ VAR arr1(),...,arrn(),  
r1[,...,rn])

or

DEFFN name(v1[,...,vn]

*name:* name of procedure or function

*v1,...,vn:* aexp, sexp; 'Call by Value'  
variables

*arr1(),...arrn():* array pointers

*r1,...,rn:* 'Call by Reference' variables

**Explanation:** There are three types of subroutine variables:

1. The variables declared in and by the subroutine itself.
2. The variables that originate from the subroutine caller.
3. The variables which are effective throughout the whole program (global variables).

It's self-evident that the changes to the first two types of variables are only possible within the subroutine or from the calling program.



The variables declared by the subroutine itself are only valid within this subroutine and are known as local variables. They are declared using the LOCAL command.

In theory, it's possible to declare variables in a subroutine and have them remain valid for the calling routine as well. However, this makes the whole program less modular, as one must keep track of where each variable was declared. It makes much more sense to declare such variables in the main program and initialize them to 0 (in case of numeric variables) or to "" (in case of string variables). The actual value assignments will then follow in the subroutines.

The variables which originate from the caller are passed to the subroutine as parameters.

These comprise the so-called 'Call by Value' and 'Call by Reference' variables. With 'Call by Value' variables the subroutine receives only the contents, i.e. the value of a variable. In place of these values the parameter list of the PROCEDURE or. FUNCTION declaration must contain the so-called "dummy" variables. The subroutines handle these as local variables. In an actual program they may, for example, appear as follows:

```
GOSUB test(5,a,"Hello",3*PI/2)
```

```
.  
PROCEDURE test(dummy%,dummy1,dummy$,dummy2)
```

```
.  
RETURN
```

or

```
FN test(5,a,"Hello",3*PI/2)
.
FUNCTION test(dummy%,dummy1,dummy$,dummy2)
.
    RETURN exp
ENDFUNC

or

FN test(5,a,b%)
.
DEFFN(dummy1,dummy2,dumm%)=(dummy1+dummy2)/dummy%
```

With 'Call by Reference' variables the subroutine receives a pointer to a variable. Because of this, any changes made to these variables by the subroutine remain valid for the caller program as well. For 'Call by Reference' variables the PROCEDURE or FUNCTION declaration must list the variable names again. These can either be identical or different from the ones defined in the call. The passing of variables as 'Call by Reference' has two advantages. One, these variables can be handled using names which are different from the ones in the caller program. Two, if these variables are declared as LOCAL in a subroutine, they can be passed to and shared with yet another subroutine.

In the parameter list of a PROCEDURE or FUNCTION, 'Call by Reference' variables must be declared after the 'Call by Value' variables. In addition, the first 'Call by Reference' variable in the parameter list must always have a VAR in front of it. This is because the 'Call by Reference' variables are always passed using pointers, which is exactly how the arrays are passed.

The one-line (DEFFN) functions cannot receive VAR parameter - that is to say no 'Call by Reference' - variables.

**Example:**

```
DATA 1,2,3,4,5,6,7,7,6,5,4,3,2,1
```

```
n%=14
```

```
DIM a(n%)
```

```
FOR i%=1 TO n%
```

```
  READ a(i%)
```

```
NEXT i%
```

```
error_code=1.333E-150
```

```
xq=0
```

```
GOSUB am(n%,a()),xq
```

```
IF xq <> error_code
```

```
  PRINT " arithmetic mean : ";xq
```

```
ELSE
```

```
  PRINT " The calculation is not possible! "
```

```
ENDIF
```

```
END
```

```
PROCEDURE am(d% VAR vector(),rv)
```

```
  LOCAL i%
```

```
  IF d%=0
```

```
    rv = error_code
```

```
  ELSE
```

```
    FOR i%=1 TO d%
```

```
      ADD rv,vector(i%)
```

```
    NEXT i%
```

```
    DIV rv,d%
```

```
  ENDIF
```

```
RETURN
```

```
// In this example a procedure am is called using a  
// GOSUB. As parameters the procedure receives the  
// value of n%, a pointer to the array a() and the  
// variable xq. In the parameter list of the proce-  
// dure am the dummy variable d% is declared for the  
// value of n% ('Call by Value' Variable). In addi-
```

## Commands and functions

---

```
// tion, there is a VAR, since only a pointer to the
// array a() and the variable xq follow. In the sub-
// routine the array a() is called vector() and the
// variable xq becomes rv.
// Using a FUNCTION declaration the above example
// appears as follows:
```

```
DATA 1,2,3,4,5,6,7,7,6,5,4,3,2,1
n%=14
DIM a(n%)
FOR i%=1 TO n%
    READ a(i%)
NEXT i%
error_code=1.333E-150
xq=FN am(n%,a())
IF xq <> error_code
    PRINT " arithmetic mean : ";xq
ELSE
    PRINT " The calculation is not possible! "
ENDIF
END
FUNCTION am(d% VAR vector())
    LOCAL i%,s
    IF d%=0
        RETURN error_code
    ENDIF
    FOR i%=1 TO d%
        ADD s,vector(i%)
    NEXT i%
    RETURN DIV s,d%
ENDFUNC
```

## PAUSE Command

**Action:** interrupts a program.

**Syntax:** PAUSE *n*  
*n*: *iexp*

**Abbreviation:** pa *n*

**Explanation:** PAUSE *n* interrupts a program for  $n/18.2$  seconds. In contrast to a FOR...NEXT waiting loop, this command is independent of the processor frequency.

**Example:** PAUSE 182 // a ten second pause  
// (a coffee break at GFA!)

**Remarks:** -

**See**

**Also:** DELAY

## PBOX Graphic command

**Action:** draws a filled rectangle.

**Syntax:** PBOX x1,y1,x2,y2  
*x1,y1,x2,y2: iexp*

**Abbreviation:** pb x1,y1,x2,y2

**Explanation:** PBOX x1,y1,x2,y2 draws a filled rectangle with the diagonally opposite corner coordinates at x1,y1 (upper left) and x2,y2 (lower right).

**Example:** CLS  
DEFFILL 5  
PBOX 10,10,100,100

**Remarks:** The fill pattern is defined using DEFFILL.

**See**  
**Also:** BOX, RBOX, PRBOX

## PCIRCLE Graphic command

**Action:** draws a filled circle.

**Syntax:** PCIRCLE *x,y,r[w1,w2]*  
*x,y,r,w1,w2: iexp*

**Abbreviation:** pci *x,y,r[w1,w2]*

**Explanation:** PCIRCLE *x,y,r[w1,w2]* draws a filled circle with the radius *r* around the centre point at the coordinates *x,y*. The optional start (*w1*) and end (*w2*) angles are used to define an arc segment. The angles *w1* and *w2* are given in whole degree steps.

**Example:**

```
CLS
DEFFILL 5
PCIRCLE 100,100,20,90,180
```

**Remarks:** The fill pattern is defined using DEFFILL.

**See**

**Also:** CIRCLE, ELLIPSE, PELLIPSE

## PEEK() Function

**Action:** reads a byte (8 bits) from an address.

**Syntax:** PEEK(addr)  
*addr: address*

**Explanation:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

**Example:**

```
PRINT PEEK(*a) // prints the first byte of
//           address of a
PRINT PEEK($40:$1E) // reads a byte at offset
//           $1E in segment $40
```

**Remarks:** BYTE{} is synonymous with PEEK() and can be used instead.

addr: see the {} function.

**See**

**Also:** BYTE{}, DPEEK(), WORD{}, INT{}, LPEEK(), LONG{}



## PEEK\$() Function

**Action:** writes the contents of memory to a string variable.

**Syntax:** `a$ = PEEK$(addr,len)`  
*a\$:* svar  
*addr:* address  
*len:* iexp

**Abbreviation:** -

**Explanation:** `a$ = PEEK$(addr,len)` writes the contents of memory at address `addr`, with the length `len`, to the variable `a$`.

**Example:** `a$=PEEK$(_TS 0,4000) // writes the screen to`  
`// variable a$.`

**Remarks:** The memory read into a variable with `PEEK$()` can again be written out with `POKE$`.  
`addr`: see the `{}` function.

**See**

**Also:** `POKE$, TGET, TPUT`

## PEEKEVENT Command

**Action:** monitors menu and window events

**Syntax:** PEEKEVENT

**Abbreviation:** peekev

**Explanation:** PEEKEVENT monitors the occurrence of events in menu bars, pop-up menus and windows. In contrast to ON MENU, no PROCEDURE is invoked by PEEKEVENT.

The relevant tests must be performed by the programmer. In contrast to GETEVENT, PEEKEVENT does not wait.

**Example:**

```

DIM m$(20)
DATA Lissajous , Figure 1 , Figure 2 , Figure 3 ,
Figure 4
DATA End ,"" , Names , Robert , Piere , Gustav
DATA Emile , Hugo ,!!
i%=-1
DO
    i%++
    READ m$(i%)           // read in the menu
                           // entries
    LOOP UNTIL m$(i%)="!!" // marks the end
    m$(i%)=""             // terminates a menu
    //
    SCREEN 16             // turns EGA mode on
    COLOR 8
    PBOX 0,0,639,349
    MENU m$()             // activates the menu bar
    //
DO
    PEEKEVENT
    
```

```
SWITCH MENU(1)
CASE 0          //  nothing happened
CASE 1          //  keypress
CASE 2,3        //  mouse click
CASE 4 TO 19    //  something to do with
//              windows
CASE 20         //  menu selection
CASE 21         //  redraw
ENDSWITCH
LOOP
```

**Remarks:** -

**See**

**Also:** ON MENU, GETEVENT, MENU()

## PELLIPSE Graphic command

**Action:** draws a filled ellipse.

**Syntax:** PELLIPSE *x,y,rx,ry*[*w1,w2*]  
*x,y,rx,ry,w1,w2:* *iexp*

**Abbreviation:** pel *x,y,rx,ry*[*w1,w2*]

**Explanation:** PELLIPSE *x,y,rx,ry*[*w1,w2*] draws a filled ellipse with the horizontal radius *rx* and the vertical radius *ry*, around the centre point with coordinates *x,y*.

The optional start (*w1*) and end (*w2*) angles are used to define an ellipsoid arc segment. The angles *w1* and *w2* are given in whole degree steps.

**Example:** CLS  
DEFFILL 5  
ELLIPSE 320,200,200,100,90,180

**Remarks:** The fill pattern is defined using DEFFILL.

**See**

**Also:** CIRCLE, PCIRCLE, ELLIPSE

## PI Variable

**Action:** contains the constant (3.14...).

**Syntax:** `x = PI`  
`x:` *aexp*

**Explanation:** -

**Example:** `PRINT PI` // prints 3.14159265359

**Remarks:** -

**See**

**Also:** -

# PLOT Graphic command

**Action:** draws a point on the screen.

**Syntax:** PLOT x,y  
x,y: *iexp*

**Abbreviation:** pl x,y

**Explanation:** PLOT x,y draws a point with coordinates x,y on the screen. The origins of the coordinate system are either in the upper left corner of the screen or window, or at the position selected with CLIPOFFSET.

**Example:**

```
SCREEN 16          EGA mode
CLS
DO
    MOUSE mx%,my%,mk%
    IF mk% AND 1
        PLOT mx%,my%
    ENDIF
LOOP

// An infinite loop which draws a point at the
// current mouse position after each mouse button
// click.
```

**Remarks:** -

**See**

**Also:** DRAW

## POINT Graphic command

**Action:** returns the colour of a point.

**Syntax:** POINT(x,y)  
*x,y: iexp*

**Explanation:** POINT(x,y) returns the colour of a point with the coordinates x,y.

**Example:** PLOT 100,100  
PRINT POINT(100,100)

**Remarks:** -

**See**  
**Also:** -

## POKE Command

**Action:** writes a byte (8 bits) to an address.

**Syntax:** POKE addr,m  
*addr: address*  
*m: iexp*

**Abbreviation:** -

**Explanation:** Writes a byte (8 bits) to an address.

**Example:** POKE(\$40:\$1E)=128 // writes 128 at offset \$1E  
// in segment \$40

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

BYTE{ } = is synonymous with POKE() and can be used instead.

addr: see the { } function.

**See**

**Also:** BYTE{ } =, DPOKE(), WORD{ } =, INT{ } =, LPOKE(), LONG{ } =



## POKE\$ Command

**Action:** writes to memory from a string variable.

**Syntax:** POKE\$ addr,a\$  
*a\$: sexp*  
*addr: address*

**Abbreviation:** -

**Explanation:** POKE\$ addr,a\$ writes the contents of memory, previously read with PEEK\$() into the variable a\$, back to address addr.

**Example:**

```
SCREEN 3
FOR i%=1 TO 500
  PRINT i%'
NEXT i%
a$=PEEK$($B800:0,4000)
CLS
REPEAT
UNTIL LEN(INKEY$)
POKE$ $B800:0,a$
```

```
// Fills the screen with numbers, erases it and waits
// for a keypress. The original screen is then
// restored.
```

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset or a 32 bit integer value is given, where the 16 high bits represent the segment and the low 16 bits the offset.

addr: see the {} function.

**See**

**Also:**

PEEK\$, TGET, TPUT

## POLYFILL Graphic command

**Action:** fills a polygon with a pattern.

**Syntax:** POLYFILL *n*,*x()*,*y()* [OFFSET *x0*,*y0*]  
*n*,*x0*,*y0*: *iexp*  
*x()*,*y()*: *avar arrays*

**Abbreviation:** polyf *n*,*x()*,*y()* [OFFSET *x0*,*y0*]

**Explanation:** POLYFILL *n*,*x()*,*y()* [OFFSET *x0*,*y0*] draws a polygon with *n* corners. The *x*,*y* coordinates of the corner points are in arrays *x()* and *y()*. The first corner point is defined in *x*(0),*y*(0) and the last in *x*(*n*-1),*y*(*n*-1). The first and last corner points are automatically connected. Optionally, a horizontal and/or vertical offset (*x0* or *y0*) can be added to these coordinates.

**Example:**

```
DIM x%(3),y%(3)
DATA 120,120,170,170,70,170,120,120
FOR i%= 0 TO 3
  READ x%(i%),y%(i%)
NEXT i%
DEFFILL 1,2,4
POLYFILL 3,x%(),y%() OFFSET -50,-50

// Draws a filled triangle.
```

**Remarks:** The colour and pattern type are defined using DEFFILL.

**See**  
**Also:** POLYLINE

### POLYLINE Graphic command

**Action:** draws connected lines with an arbitrary number of corners.

**Syntax:** POLYLINE *n*,*x()*,*y()* [OFFSET *x0*,*y0*]  
*n*,*x0*,*y0*: *iexp*  
*x()*,*y()*: *avar arrays*

**Abbreviation:** *polyl* *n*,*x()*,*y()* [OFFSET *x0*,*y0*]

**Explanation:** POLYLINE *n*,*x()*,*y()* [OFFSET *x0*,*y0*] draws connected lines with *n* corners. The *x*,*y* coordinates of the corner points are in arrays *x()* and *y()*. The first corner point is defined in *x*(0),*y*(0) and the last in *x*(*n*-1),*y*(*n*-1). The first and last corner points are automatically connected. Optionally, a horizontal and/or vertical offset (*x0* or *y0*) can be added to these coordinates.

**Example:**

```
DIM x%(3),y%(3)
DATA 120,120,170,170,70,170,120,120
FOR i%= 0 TO 3
    READ x%(i%),y%(i%)
NEXT i%
POLYLINE 4,x%(),y%() // draws a triangle
```

**Remarks:** -

**See**

**Also:** POLYFILL

## POPUP() Function

**Action:** creates a pop-up menu.

**Syntax:**       ~POPUP(a\$,x,y,i)   or  
                 a=POPUP(a\$,x,y,i)  
          a\$:       sexp  
          a,x,y,i:   iexp

**Explanation:**   ~POPUP(a\$,x,y,i) or a=POPUP(a\$,x,y,i) create a pop-up menu in graphic mode. The parameter a\$ is a string expression which contains the pop-up menu entries. The individual entries are separated from each other by a vertical bar "|". The first entry specifies the non-selectable title of the pop-up menu. All remaining entries are selectable. The selection is performed with the mouse or by using the cursor keys and Return.

In addition, the so called "hotkeys" can also be used when the designated key is prefixed by an underdash "\_". x and y specify the coordinates of the upper left corner of the pop-up menu relative to the upper left corner of the screen. i is a flag with the following meaning:

i = 1	The pop-up menu entry will be centred.
i = 2	The pop-up menu entry will be left justified.
i = 3	The pop-up menu does not appear at the location specified in the x and y coordinates, but at the current mouse position.

## Commands and functions 3.892

---

The POPUP function call returns a value which contains the number of the selected entry. The value returned by the function can be obtained in two ways:

```
a%=POPUP(a$,x,y,i)
```

Following an entry selection, a% contains the number of the selected entry, and/or

```
~POPUP(a$,x,y,i)
```

which fills the menu array MENU(). By using MENU(1)=20 it can then be determined whether a POPUP entry was selected. If it was, MENU(0) contains the number of the selected entry.

### Example:

```
SCREEN 16 // EGA mode
a$="TITLE|Entry _1|Entry _2|Entry _3"
a%=POPUP(a$,100,100,1)
PRINT a%
REPEAT
UNTIL LEN(INKEY$)
```

```
// This program turns on the EGA mode and displays a
// POPUP menu with the title "TITLE" and entries
// "Entry 1", "Entry 2" and "Entry 3". The number of
// the entries are also declared as "hotkeys". After
// the selection of an entry its number is displayed
// and the program waits for a keypress.
```

The same program can also be written like this:

```
SCREEN 16           // EGA mode
a$="TITLE|Entry _1|Entry _2|Entry _3"
~POPUP(a$,100,100,1)
DO
  GETEVENT
  e%=MENU(1)
  ee%=MENU(0)
  IF e%=20
    PRINT ee%
  ENDIF
LOOP UNTIL ee%
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

**Remarks:**

The advantage of evaluating the pop-up menus with GETEVENT, PEEKEVENT or ON MENU is that a program, which contain both a menu bar and pop-up menus, only needs one evaluation routine.

**See**

**Also:**

MENU()

### POS() Function

**Action:** returns the number of characters printed since the last carriage return.

**Syntax:** POS(*n*)  
*x*: *aexp*; *dummy*

**Abbreviation:** -

**Explanation:** returns the number of characters printed on the screen since the last carriage return. POS() internally performs an AND 255. The parameter *x* is a dummy.

The value returned by POS() should not be confused with the current cursor position. Let's take a screen with 80 characters per line. If a string with 120 characters is printed from the start of the first line the cursor will end up in column 40 on the 2nd line. POS(), however, will return the value 120.

The control characters are not counted by POS() but they can influence the value returned by it. For example, CHR\$(13) = the carriage return sets the counter to zero while CHR\$(8) = the backspace decreases the counter by 1.

**Example:** PRINT POS(0)

```
// Returns the number of characters printed since the  
// last CR.
```

**Remarks:** -

**See**

**Also:** CRSCOL, CRSLIN



## PRBOX Graphic command

**Action:** draws a filled rectangle with rounded corners.

**Syntax:** PRBOX x1,y1,x2,y2  
*x1,y1,x2,y2: iexp*

**Abbreviation:** prb x1,y1,x2,y2

**Explanation:** PRBOX x1,y1,x2,y2 draws a filled rectangle with the diagonally opposite corner coordinates at x1,y1 (upper left) and x2,y2 (lower right). The rectangle corners are rounded.

**Example:** CLS  
DEFFILL 5  
PRBOX 10,10,100,100

**Remarks:** The fill pattern is defined using DEFFILL.

**See**

**Also:** BOX, RBOX, PBOX

### PRED() Function

**Action:** calculates the first natural number smaller than an integer expression.

**Syntax:** PRED(*n*)  
*n*: *iexp*

**Explanation:** PRED(*n*) returns the first natural number smaller than the integer expression *n*.

**Example:**

```
PRINT PRED(4*11-1)    // prints    42
1%=PRED(4*11-1)
PRINT 1%              // prints    42
```

**Remarks:** -

**See**

**Also:** ADD(), SUB(), MUL(), DIV(), MOD(), SUCC()

## PRED() String function

**Action:** returns a character whose ASCII value is one less than the first character of a string expression.

**Syntax:** PRED(a\$)  
*a\$: sexp*

**Abbreviation:** -

**Explanation:** -

**Example:** PRINT PRED("Hello World") // prints G

**Remarks:** PRED(a\$) corresponds to CHR\$(PRED(ASC(a\$))).

**See**

**Also:** SUCC()

# PRINT Command

**Action:** output of numerical and/or string expressions

**Syntax:** PRINT x[,y,a\$,...]  
*x,y:* *aexp*  
*a\$:* *sexp*

**Abbreviation:** p

**Explanation:** A PRINT without any parameters performs a line feed. If the cursor is on the last line, the whole screen is moved up by one line. A PRINT followed by an expression prints this expression at the current cursor position.

PRINT AT, LOCATE, VTAB and HTAB can be used to position the cursor. The strings must be enclosed in quotation marks.

PRINT can be followed by several (different) expressions which must be separated by a comma, a semi-colon or an apostrophe.

The comma moves the cursor to the next tab position - a column fully divisible by 16. When the last column is reached the cursor is moved to column 17 on the next line. The semi-colon performs the output of expressions without any spaces between them. The apostrophe, however, inserts a space between the expressions.

A line feed is performed after each PRINT except when the last expression is followed by a semi-colon. In such a case the next PRINT output resumes from the end of the previous one.

**Example:** PRINT 3\*4+12 // prints 24

PRINT "3\*4+12 = ";3\*4+12 // prints 3\*4+12 = 24

```
a$="GFA Systemtechnik"  
b$="BASIC"  
PRINT LEFT$(a$,4)+b$      // prints  GFA BASIC  
  
PRINT "A"'CHR$(66)'"C"  // prints  A B C
```

**Remarks:**

-

**See**

**Also:**

PRINT AT, PRINT USING, PRINT AT...USING,  
PRINT #, PRINT # USING

### PRINT # Command

**Action:** outputs data to a file channel

**Syntax:** PRINT #*n*,*x*  
*n*: *iexp*; *channel number*  
*x*: *aexp* or *sexp*

**Abbreviation:** p #*n*,*x*

**Explanation:** PRINT #*n*,*x* outputs data to a previously opened channel. *n* is this channel number in the range from 0 to 99. Other than that PRINT # is equivalent to PRINT.

**Example:**

```
OPEN "o",#1,"A:\TEST.DAT"  
a$="Hello GFA"  
PRINT #1,a$  
PRINT #1,"GFA-","BASIC"  
CLOSE #1
```

```
// Opens the file TEST.DAT on drive A and writes the  
// strings Hello GFA, GFA- and BASIC to it.
```

**Remarks:** -

**See**

**Also:** PRINT, PRINT USING, PRINT #, USING, WRITE, WRITE #

## PRINT AT Command

**Action:** output of numerical and/or string expressions

**Syntax:** PRINT AT(column,row);x,[AT(column,row);y,...]  
*column:*      *ivar; cursor column*  
*row:*          *ivar; cursor row*  
*x,y:*          *aexp or sexp*

**Abbreviation:** p at(...)

**Explanation:** PRINT AT(column,row) followed by an expression, performs the output of this expression at the cursor position defined by column and row. PRINT AT() without any parameters performs a line feed. The list of parameters after PRINT AT() can contain other AT() instructions which then apply to printing of expressions following after them. i.e. at the corresponding column and row.

For further clarification see PRINT.

**Example:** a\$="GFA Systemtechnik"  
b\$="BASIC"  
PRINT AT(4,6);LEFT\$(a\$,4)+b\$

// Prints GFA BASIC in fourth column of the sixth row

```
PRINT AT(4,6);"What do you get";AT(4,7);"when you  
multiply"  
PRINT AT (4,8);"6 by 9";AT(4,10);" 42!!! "
```

**Remarks:** -

**See**

**Also:** PRINT, PRINT USING, PRINT AT...USING, PRINT  
#, PRINT # USING

## PRINT AT ... USING Command

**Action:** formatted output to screen

**Syntax:** PRINT AT(column,row);USING format\$,a[;]  
*column,row: ivar*  
*format\$: sexp*  
*a: exp or sexp*

**Abbreviation:** p at();using ...

**Explanation:** In principle, PRINT AT ... USING is equivalent to PRINT AT, except that the expression to be printed is formatted first using the format\$ template.

For description of formatting template refer to PRINT USING.

**Example:** f1\$="#.###"  
PRINT AT(4,6);USING f1\$,PI// prints 3.142 in the  
// sixth column on the  
// fourth line

**Remarks:** The decimal point and comma can be swapped using the MODE command.

**See**

**Also:** STR\$(), PRINT AT()



## PRINT USING Command

**Action:** formatted output to screen

**Syntax:** PRINT USING format\$,a[;]  
*format\$:*       *sexp*  
*a:*               *aexp or sexp*

**Abbreviation:** p using ...

**Explanation:** In principle, PRINT USING is equivalent to PRINT, except that the expression to be printed is formatted first using the format\$ template.

The following characters are available for formatting of numerical expressions:

- # place holder for a digit. When this digit is the last digit in the format template it is rounded off before output.
- . used to indicate the decimal point in between the # characters.
- , inserts a comma at the corresponding place between the # characters and can, for example, be used to separate the thousands.
- reserves a place for the minus sign. If the number is positive a space is printed instead. This format character is only allowed before or after the formatting template.
- + similar to the - character only a plus sign is displayed before of after a positive number. The plus and the minus characters cannot be combined.
- \* is an alternative to #, the leading zeros are replaced by spaces.

## Commands and functions

---

\$ when placed immediately before the very first #, it performs the printing of a \$ sign in front of the number.

^ sets the exponential format (E+000). In this format the # character specifies the length of the mantissa, while the ^ character specifies the length of the exponents including the E+ or E-. If there are several # characters before the decimal point, the exponent is adjusted so that it's divisible by the count of these characters. The negative numbers must contain the sign character.

The following characters are available for formatting of string expressions:

& performs the output of the whole string.

! limits the output to the first character in the string.

\..\ specifies the number of characters to be printed from a string. The count includes both \ characters.

\_ an underline performs the output of the next character in template as a literal.

### Example:

```
f1$="#.###"
f2$="#.###_._._"
f3$="\...\\"
//
PRINT USING f1$,PI           // prints  3.142
PRINT USING f2$,PI           // prints  3.142...
PRINT USING f3$,"Hello GFA"  // prints  Hello

PRINT USING f1$,PI;f2$,PI;f3$,"Hello GFA"

// prints
// 3.142 3.142... Hello

PRINT USING "###.###^10",2^10 // prints  1.024E+03
```

**Remarks:** The decimal point and comma can be swapped using the **MODE** command.

**See**  
**Also:** **STR\$()**, **PRINT**

### PRINT #n,USING Command

**Action:** output of formatted data to a file channel

**Syntax:** PRINT #n,USING format\$,x  
*n:* *iexp; channel number*  
*format\$:* *sexp*  
*x:* *aexp or sexp*

**Abbreviation:** p #n,USING...

**Explanation:** In principle, PRINT # USING is equivalent to PRINT #, except that the expression to be printed is formatted first using the format\$ template. For characters used for formatting of expressions refer to the PRINT USING command.

**Example:**

```
f1$="#.###"  
f2$="#.###_._._."  
f3$="\...\\"  
OPEN "o",#1,"A:\TEST.DAT"  
PRINT #1,USING f1$,PI           // prints 3.142  
PRINT #1,USING f2$,PI           // prints 3.142...  
PRINT #1,USING f3$,"Hello GFA" // prints Hello  
PRINT #1,USING "###.###^",2^10 // prints 1.024E+03  
CLOSE #1
```

**Remarks:** -

**See**

**Also:** PRINT, PRINT USING, PRINT #, WRITE, WRITE #

## PROCEDURE...RETURN Structure

**Action:** gathers together into a subroutine, program segments which are used repeatedly within the program.

**Syntax:**

```
PROCEDURE name [(v1,...vn,VAR  
arr1(),...,arrn(),r1,...,rn)]  
    LOCAL l1,...,ln  
    // programsegment
```

<i>RETURN name:</i>	<i>subroutine name</i>
<i>v1,...,vn:</i>	<i>Call by Value variables</i>
<i>arr1(),...,arrn():</i>	<i>arrays</i>
<i>r1,...,rn:</i>	<i>Call by Reference variables</i>
<i>l1,...,ln:</i>	<i>local variables</i>

**Abbreviation:**

```
proc Name [...]  
    loc ...  
ret
```

**Explanation:** In addition to FUNCTIONS, PROCEDURES are the most important components of structured programming. They allow a program to be broken up into modules which are then called to perform a particular task.

The call of a PROCEDURE from a program is performed with GOSUB procedurename, @procedurename or simply with procedurename.

The body of a procedure is composed of the declaration (PROCEDURE), subroutine name, parameter list, local variable definition (LOCAL...), subroutine program statements and procedure end (RETURN).

The statement **PROCEDURE test** [parameterlist] declares a subroutine called "test" and, optionally a parameter list. The parameter list must start with a list of variables - separated by commas - which are to be passed to the subroutine as 'Call by Value' variables (see Parameter list).

Should the list also contain additional arrays and/or 'Call by Reference' variables, the last 'Call by Value' variable must be followed by a comma and a **VAR**. This is then followed by the names of the arrays with brackets and 'Call by Reference' variables.

The declaration line is followed by a program line with definitions of local variables (see Parameter list). This line begins with the **LOCAL** command.

Should a procedure need several local variables, they are all listed after **LOCAL** and separated by commas. Several **LOCAL** lines are allowed.

The definition of local variables is followed by the program lines that comprise the subroutine. The subroutine must end with a **RETURN** command. This indicates the end of **PROCEDURE** declaration and the return to the program statement immediately after the calling program statement.

### Example:

```
Do
  INPUT "Enter text ";a$
EXIT IF a$=""
  GOSUB center_text(a$) // PROCEDURE-call
LOOP
END
PROCEDURE center_text(a$)
  LOCAL l%,x_pos%
  l%=LEN(a$)
```

```
x_pos%=DIV(SUB(80,1%),2)
LOCATE 5,x_pos%
PRINT a$
RETURN
```

```
// The program waits in an endless loop for you to to
// enter some text. If you don't input any text, the
// program is terminated.
// Otherwise, the text is passed to the subroutine
// 'center_text', which - based on a resolution of 80
// characters per line - displays the centred text on
// line 5.
```

**Remark:** Unlike FUNCTIONS, PROCEDURES can't be included in an arithmetic or string expression!

**See**

**Also:**

```
FUNCTION...name...ENDFUNC,
FUNCTION...name$...ENDFUNC
```

## PSAVE Command

**Action:** saves the current GFA-BASIC program with LIST protection.

**Syntax:** PSAVE a\$  
a\$: *sexp*

**Abbreviation:** psa

**Explanation:** PSAVE a\$ can only be performed in direct mode. PSAVE must be followed by the name of the file under which the current program should be saved. Invoking this command saves the program LIST protected. This means that after it has been reloaded using LOAD, the program can no longer be displayed.

**Example:** PSAVE"C:\TEST.GFA"

```
// Saves the current program under the name TEST.GFA
// in partition C of the hard drive with LIST pro-
// tection.
```

**Remarks:** -

**See Also:** SAVE, LIST



## PSET Command

**Action:** sets a graphic point.

**Syntax:** PSET *x,y,c*  
*x,y,c: iexp*

**Abbreviation:** -

**Explanation:** PSET *x,y,c* sets a graphic point at the coordinates *x* and *y* in colour *c*. PSET can be used as an alternative to:

```
COLOR c  
PLOT x,y
```

however, it will not change the current colour.

**Example:**

```
SCREEN 16  
DO  
    PSET RAND(_X),RAND(_Y),RAND(_C)  
LOOP
```

```
// Fills the screen slowly with many multicoloured  
// points.
```

**Remarks:** -

**See**

**Also:** COLOR, PLOT

# PUT Command

**Action:** copies a portion of the screen saved with GET to screen memory.

**Syntax:** PUT *x,y*,screensegment\$[,mode]  
*x,y,mode:* *iexp*  
*screensegment\$:* *svar*

**Abbreviation:** -

**Explanation:** PUT *x,y* copies a portion of the screen saved with GET back to screen memory, so that the upper left corner of the segment is aligned with the *x,y* coordinates on the screen.

The optional mode parameter defines how will the bit-pattern in screensegment\$ be combined with the current screen contents. The mode parameter has the same values as in GRAPHMODE. This means:

n	Rule	Effect
1	set	The new bit pattern is moved to the current screen.
2	OR	All points which are set in, either the current screen or the new pattern, are set.
3	XOR	Only the points which are different in both the current screen and the new pattern are set.
4	AND	All points which are set in both the current screen and the new pattern are set.

**Example:**

```
DEFFILL 2
PBOX 10,10,20,20
GET 10,10,20,20,a$
REPEAT
    MOUSE mx%,my%,mk%
    IF MOUSEK AND 1
        PUT mx%,my%,a$
    ENDIF
UNTIL MOUSEK AND 2

// Draws a filled rectangle and saves it in the
// variables a$.
// When the left mousebutton is pressed, the rec-
// tangle is moved to the current mouse position on
// the screen.
```

**Remarks:**

-

**See****Also:**

GET

### PUT # Command

**Action:** writes a record to a random access file.

**Syntax:** PUT #n[,record]  
*n:* *iexp; channel*  
*record:* *iexp*

**Abbreviation:** -

**Explanation:** PUT # writes a record to an R-file through the channel n (from 0 to 99), previously opened with OPEN. record is an optional parameter and contains a value between 1 and the number of records within the file. If record is not specified the next record in file is always written out. If record is specified it is written out.

**Example:**

```
OPEN "R",#1,"A:\Adresses.DAT",62
FIELD #1,24 AS name$,24 AS str$,2 AT(*postcode&),12
AS city$
//
FOR i%=1 TO 5
    INPUT "NAME      : ";n$
    INPUT "Street    : ";s$
    INPUT "Postcode: ";postcode&
    INPUT "City      : ";c$
    LSET name$=n$
    LSET str$=s$
    LSET city$=c$
    PUT #1,i%
CLS
NEXT i%
CLOSE #1
//
OPEN "R",#1,"A:\Adresses.DAT",62
FIELD #1,24 AS name$,24 AS str$,2 AT(*postcode&),12
AS city$
```

```
//  
FOR i%=1 TO 5  
  GET #1,i%  
  PRINT "Record number: ";str$(i%,3)  
  PRINT "NAME      : ";name$  
  PRINT "Street   : ";str$  
  PRINT "Postcode: ";postcode&  
  PRINT "City     : ";city$  
NEXT i%  
CLOSE #1  
  
// A channel for the random access file is opened  
// first.  
// Next, the record is divided with FIELD into: 24  
// bytes for the name, 24 bytes for the street, two  
// bytes for the postal code and 12 bytes for the  
// city, which all together totals 62 bytes. The  
// FOR...NEXT loop writes five records to the file  
// ADDRESSES.DAT on drive A.  
// And finally, these records are read in using GET  
// and displayed on the screen again.
```

**Remarks:** PUT # can only add one record to a file. To add several records to an R-file a loop containing a PUT # must be created.

**See**

**Also:**

FIELD, GET #, RECORD #

## QSORT Function

**Action:** sorts the elements in an array by its size using the Quicksort method.

**Syntax:**

```
QSORT x(s)[,n][,m%()] or
QSORT x$(s) WITH n()[,n[,m%()]]
s:          + or - for ascending or descending
            order
n:          iexp
x():        one dimensional floating point array
x$():       one dimensional string array
n():        one dimensional integer array with 1,2 or 4
            byte variables
m%():       one dimensional integer array with 4 byte
            variables
```

**Abbreviation:** -

**Explanation:** The s enclosed in round brackets can be replaced with a "+", a "-" or may be left out. "+" or no specification results in arrays y() and m%() being sorted in ascending order. In this case, after the sorting, the smallest array element assumes the smallest index (0 for OPTION BASE 0 or 1 for OPTION BASE 1). "-" results in the array being sorted in descending order. In this case, after the sorting, the biggest array element assumes the smallest index.

The parameter n specifies that only the first n elements of the array should be sorted. For OPTION BASE 0 these are the elements with indices 0 to "n"-1, and for OPTION BASE 1 the elements with indices 1 to "n". If n is given explicitly, it can be followed by a LONG integer array, which will be sorted together with the array a(), that is to say, each swap in array a()

will also be performed in array `m%()`. This is particularly useful when the sorted array `a()` contains the sort key (for example the postal code), while other arrays contain the additional information whose information must maintain the same order as the keys.

When sorting string arrays (`x$()`) a sort criterion can be specified with `WITH`, in form of an array with at least 256 elements. If `WITH` is not given the normal ASCII table is used as the sort criterion.

### Example:

```
OPTION BASE 1
a:
DATA 10,-3,5,21
c:
DATA "A","B","C","D"
d:
DATA "Who","How","What","Where"
DIM a(4),b%(4),c$(4),d$(4)
RESTORE a
MAT READ a()
RESTORE c
FOR i%=1 TO 4
  READ c$(i%)
  b%(i%)=i%
NEXT i%
RESTORE d
FOR i%=1 TO 4
  READ d$(i%)
NEXT i%
FOR i%=1 TO 4
  PRINT STR$(a(i%),5,2)''
  PRINT STR$(b%(i%),5,2)''
  PRINT c$(i%)''
  PRINT d$(i%)
NEXT i%
PRINT
QSORT a(),4,b%()
```

## Commands and functions

---

```
FOR i%=1 TO 4
  PRINT STR$(a(i%),5,2)''
  PRINT STR$(b%(i%),5,2)''
  PRINT c$(b%(i%))''
  PRINT d$(b%(i%))
NEXT i%
```

```
// Prints first of all (unsorted)
// 10.00 1.00 A Who
// 3.00 2.00 B How
// 5.00 3.00 C What
// 20.00 4.00 D Where
// and then (sorted)
// 3.00 2.00 B How
// 5.00 3.00 C What
// 10.00 1.00 A Who
// 20.00 4.00 D Where
```

**Remarks:** -

**See**

**Also:** SSORT



## QUIT Command

**Action:** terminates a GFA-BASIC program and returns back to the calling program.

**Syntax:** QUIT [*i*]  
*i*: *iexp*

**Abbreviation:** -

**Explanation:** The QUIT command terminates the current GFA-BASIC program and returns to the calling program. Optionally, a 16 bit integer value can be returned to the calling program. The following convention applies:

*i* = 0 the program was executed without error.

*i* > 0 an internal program error has occurred.

*i* < 0 an operating system error has occurred.

**Example:**

```
SCREEN 16                // EGA mode
OPENW #1,10,10,200,100,-1
REPEAT
    a$=inkey$
UNTIL a$=chr$(27)
CLOSEW #1
SCREEN 3
QUIT
```

```
// Draws a window on the screen, waits for the
// pressing of the Esc key, turns the text mode on
// and returns to the calling program (operating
// system).
```

## **Commands and functions**

---

**Remarks:** SYSTEM is synonymous with QUIT and can be used instead.

**See**  
**Also:** SYSTEM, EDIT



### **RAD()** Trigonometrical function

**Action:** returns the radian equivalent of an angle specified in degrees.

**Syntax:** RAD(*x*)  
*x*: *aexp*; angle in degrees

**Explanation:** RAD(*x*) is equivalent to  $x \cdot \text{PI} / 180$ .

**Example:**

```
PRINT RAD(90)           // prints  PI/2
PRINT RAD(180)          // prints  PI
PRINT RAD(270)          // prints  3*PI/2
PRINT RAD(360)          // prints  2*PI
```

**Remarks:** RAD(*x*) is the reverse function of DEG(*x*), which means:

$$\text{RAD}(\text{DEG}(\text{PI})) = \text{PI} = 3.14\dots$$

**See**

**Also:** DEG()

## RAND() Function

**Action:** returns a 16-bit random integer number in the range from 0 to n-1 inclusive.

**Syntax:** RAND(*n*)  
*n*: *iexp*

**Explanation:** An internal 32 bit random number generator is used for calculation of random numbers. Although faster than the 48 bit random number generator used with RANDOM() or RND(), its output is "less random".

**Example:** PRINT RAND(256):  
  
// Prints a random integer number between 0 and 255.

**Remarks:** -

**See**

**Also:** RND(), RANDOM(), RANDOMIZE

## RANDOM() Function

**Action:** returns an integer random number between 0 (inclusive) and x (exclusive).

**Syntax:** RANDOM(*x*)  
*x*: *aexp*

**Explanation:** When the numeric expression *x* is an integer, all numbers have the same probability of being selected, and vice versa. RANDOM(*x*) is equivalent to TRUNC(RND\**x*).

**Example:** PRINT RANDOM(10)  
  
// Prints a random number between 0 and 10.

**Remarks:** -

**See**

**Also:** RND(), RAND(), RANDOMIZE

## RANDOMIZE Command

**Action:** seeds the random number generators.

**Syntax:** RANDOMIZE[n]  
*n: iexp*

**Abbreviation:** -

**Explanation:** RANDOMIZE[n] seeds the random number generators with the value n. If the random number generator is seeded several times with the same  $n \neq 0$ , the same sequence of "random numbers" is generated.

Every time a program is run the random number generators are seeded with a "random" number. Therefore, if RANDOMIZE is not used, each program run will result in RND, RANDOM or RAND producing different random numbers.

RANDOMIZE (without parameters) or RANDOMIZE 0 seeds the random number generator with a "random" number just like when a program starts up.

**Example:** -

**Remarks:** -

**See Also:** RND(), RANDOM(), RAND()

## **RBOX Graphic command**

**Action:** draws a rectangle with rounded corners.

**Syntax:** RBOX x1,y1,x2,y2  
*x1,y1,x2,y2: iexp*

**Abbreviation:** rb x1,y1,x2,y2

**Explanation:** RBOX x1,y1,x2,y2 draws a rectangle with the diagonally opposite corner coordinates at x1,y1 (upper left) and x2,y2 (lower right). The rectangle corners are rounded.

**Example:** RBOX 10,10,100,100

**Remarks:** -

**See**

**Also:** BOX, PBOX, PRBOX



## RC\_INTERSECT() Function

**Action:** determines the overlapping area between two rectangles.

**Syntax:** `fil = RC_INTERSECT(x1,y1,w1,h1,x2,y2,w2,h2)`  
*fil:* Boolean variable  
*x1,y1,w1,h1:* *iexp*;  
*x2,y2,w2,h2:* *var*; return values

**Explanation:** The RC\_INTERSECT() function tests if two rectangles overlap. The upper left corner of the first rectangle is specified in x1 and y1, the width in w1 and the height in h1.

The upper left corner of the second rectangle is specified in x2 and y2, the width in w2 and the height in h2. If the two rectangles overlap the function returns TRUE (-1), otherwise it returns a FALSE (0).

The upper left corner of the overlapping area between the two rectangles is returned in x2 and y2, the width in w2 and the height in h2. Because of this the last four parameters in the RC\_INTERSECT function must always be integer variables (VAR parameter).

If the two rectangles do not overlap, the x2,y2,w2 and h2 variables contain the coordinates of a rectangle between the two given rectangles. The width and height are then either negative or 0.

The first four parameters can also be specified with expressions. The last four parameters must be given as variables. They are changed by RC\_INTERSECT.

## Commands and functions

---

### Example:

```
SCREEN 16
BOX 10,10,400,200
x%=50
y%=50
w%=400
h%=200
BOX x%,y%,add(x%,w%),add(y%,h%)
IF RC_INTERSECT(10,10,400,200,x%,y%,w%,h%)
  DEFFILL 20
  PBOX x%,y%,add(x%,w%),add(y%,h%)
ENDIF
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3

// Draws two rectangles and fills the overlapping
// area with a pattern.
```

### Remarks:

-

### See

### Also:

GETFIRST, GETNEXT

## READ Command

**Action:** reads data from DATA lines into variables.

**Syntax:** READ k1[,k2,k3,...]  
*k1,k2,k3... avar, ivar or svar*

**Abbreviation:** rea k1[...]

**Explanation:** READ reads data from DATA lines into variables. The value read is always the one pointed to by the data pointer. At the beginning of the program this data pointer points to the first value on the first DATA line. After reading this value the pointer points to the second value on the first DATA line, and so on. The data pointer can be modified by using markers and the RESTORE command. A marker is a sequence of characters which can be composed of digits, letters, underline characters and dots. Each marker must end with a colon.

**Example:**

```
DATA 1,2,3,4,5,6,7,8,9,10
FOR i%=1 TO 10
  READ a%
  PRINT a%
NEXT i%
EDIT
```

```
// Reads a value from the DATA line into variable a%
// and prints this value to the screen.
```

```
DATA 1,2,3,4,5,6,7,8,9,10
FOR i%=1 TO 4
  READ a%
  PRINT a%
NEXT i%
```

## Commands and functions

---

```
// Reads the first four values from the DATA line
// into variable a% and prints them to the screen. At
// the end of the loop the data pointer points to
// value 5, that is to say, to the 5th value on the
// DATA line.
```

```
FOR i%=1 TO 6
  READ a%
  PRINT a%
NEXT i%
```

```
// Reads the fifth to tenth value from the DATA line
// into variable a% and prints it to the screen.
```

```
1a:                                // marker
DATA 1,2,3,4,5,6,7,8,9,10
FOR i%=1 TO 4
  READ a%
  PRINT a%
NEXT i%
```

```
// Reads the first 4 values from the DATA line into
// variable a% and prints them to the screen. At the
// end of the loop the data pointer points to value
// 5, that is to say, to the 5th value on the DATA
// line.
```

```
RESTORE 1a
```

```
// The data pointer is set to the (first) DATA line
// behind the marker 1a.
```

```
FOR i%=1 TO 6  
  READ a%  
  PRINT a%  
NEXT i%
```

```
// Reads the first to sixth value from the DATA line  
// into variable a% and prints it to the screen.
```

**Remarks:**

-

**See**

**Also:**

DATA, RESTORE

### RECALL # Command

**Action:** fast load of text files saved with STORE.

**Syntax:** RECALL #*n*,a\$( ),*m*,*j*  
*n*: *iexp*; channel number  
*a\$( )*: *one dimensional string array*  
*m*: *iexp*  
*j*: *ivar*

**Abbreviation:** reca #*n*,a\$( ),*m*,*j*

**Explanation:** RECALL #*n*,a\$( ),*m*,*j* reads through an already opened channel *n* (from 0 to 99) *m* lines from a text file saved with STORE, into the string array a\$( ). If *m* is greater than the number of elements in the string array, the number of reads is automatically limited (*m* = -1 fills the whole array). If during reading an EOF is reached the reading is stopped without reporting an error. At the end of the read the variable *x* contains the number of strings actually read in.

**Example:**

```
DIM a$(1000)
OPEN "i",#1,"A:\TEST.TXT"
RECALL #1,a$( ),-1,x%
CLOSE #1
PRINT x%
```

```
// Reads the complete text file TEST.TXT on drive A
// into the string array a$( ) and, when finished,
// prints how many strings have been read in.
```

**Remarks:** RECALL is for strings what BLOAD or BGET are for general arrays and memory areas.

**See**

**Also:** STORE

## RECORD # Command

**Action:** specifies the next record to be read with GET # or written with PUT #.

**Syntax:** RECORD #*n*,*record*  
*n*: *iexp*; *channel number*  
*record*: *iexp*

**Abbreviation:** rec #*n*,*record*

**Explanation:** -

**Example:** RECORD #1,15

// In this example GET #1 would read or PUT #1 would  
// write the record No. 15

**Remarks:** -

**See**

**Also:** FIELD, GET #, PUT #, SEEK

### RELSEEK # Command

**Action:** relative positioning of the data pointer

**Syntax:** RELSEEK #*n*,count  
*n*: *iexp*; channel number  
*pos*: *iexp*

**Abbreviation:** rels #*m*,count

**Explanation:** RELSEEK #*n*,*pos* enables access to index sequential files. *n* specifies the channel number from (0 to 99) previously opened with OPEN. RELSEEK can only be used with files and not with peripheral devices. The value in count causes the pointer to move count bytes relative to the current data pointer position. When count is positive the data pointer moves towards the file end, and when count is negative the data pointer moves towards the file start. When count is given, care should be taken not to reach beyond the start or end of file.

**Example:**

```
OPEN "o",#1,"A:\TEST.DAT"
PRINT #1,STRING$(20,"*")
SEEK #1,10
OUT #1,ASC("!"),ASC("0")
RELSEEK #1,5
OUT #1,ASC("?")
RELSEEK #1,-10
OUT #1,ASC("/")
CLOSE #1
OPEN "i",#1,"A:\TEST.DAT"
LINE INPUT #1,a$
CLOSE #1
PRINT a$                // prints *****/**!0*****?*****
```



**Remarks:** RELSEEK is only applicable when used with PRINT #, OUT #, INPUT # etc.

**See**

**Also:** SEEK, LOC()

### REM Command

**Action:** program comment

**Syntax:** REM

**Abbreviation:** -

**Explanation:** When a line begins with a REM it can be followed by any comment. No syntax control is performed on this line and all subsequent characters are no longer interpreted as commands, function or variables.

**Example:**

```
SCREEN 3
FOR i%=1 TO 10
    REM FOR...NEXT loop
    // FOR...NEXT loop
    /* FOR...NEXT loop
    ' FOR...NEXT loop
    PRINT i%
NEXT i%
```

**Remarks:** //, /\* and ' are synonymous with REM and can be used instead.

**See**

**Also:** //, /\*, '

## RENAME...AS Command

**Action:** renames a file.

**Syntax:** RENAME old\$ AS new\$  
*old\$,new\$: sexp; old and new file names*

**Abbreviation:** rena old\$ as new\$

**Explanation:** -

**Example:**

```
old$="A:\TEST.TXT"
new$="A:\NEW.TXT"
IF EXIST old$
    RENAME old$ AS new$
ENDIF

// Checks if the file with the name TEST.TXT exists
// on drive A and renames it to NEW.TXT
```

**Remarks:** NAME...AS is synonymous with RENAME...AS and can be used instead.

**See**

**Also:** NAME...AS

### REPEAT...UNTIL Structure

**Action:** a terminal program loop which runs until the condition at the end of the loop is logically "true".

**Syntax:**

```
REPEAT
    // programsegment
UNTIL condition
condition: any numeric, logical or string condition
```

**Abbreviation:**

```
rep
    // programsegment
unt condition
```

**Explanation:** The end of a REPEAT...UNTIL loop must contain a numeric, logical or string condition, which is evaluated after each execution of the body of the loop. If the condition is logically "true", a branch is taken to the program statement immediately after UNTIL. Otherwise, the body of the loop is executed again.

The REPEAT...UNTIL loop is an exit tested loop. This means that the loop executes at least once and the test, whether or not, the condition is fulfilled is first performed at the end of the loop.

By using an "EXIT IF" command, the REPEAT...UNTIL loop can be terminated regardless of whether the loop condition is fulfilled.

**Example:**

```
REPEAT a$= UPPER$(INKEY$)
    // programsegment
UNTIL a$="A"

// A loop which runs until lowercase or uppercase "a"
// is entered from the keyboard.
```

**Remarks:**

-

**See**

**Also:**

FOR...NEXT, WHILE...WEND, DO...LOOP

## RESTORE Command

**Action:** positions the data pointer for READ

**Syntax:** RESTORE [marker]  
*marker:* any number or string

**Abbreviation:** rest marker

**Explanation:** RESTORE serves to position the data pointer so that the READ command can read the data from the specific DATA lines. RESTORE without a marker sets the data pointer to the first value on the first DATA line. RESTORE marker sets the data pointer to the first value on the first DATA line after the given marker.

**Example:**

```

1a:                                // marker
DATA 1,2,3
2a:                                // marker
DATA Peter,Klaus,Michael
RESTORE 1a                        // position the data
//                                pointer
FOR i%=1 TO 3
    READ a%
    PRINT a%''
NEXT i%
PRINT
//
RESTORE 2a                        // position the data
//                                pointer
FOR i%=1 TO 3
    READ a%
    PRINT a%
```

NEXT i%

EDIT

// Prints

// 1 2 3

// Peter Klaus Michael

**Remarks:**

-

**See**

**Also:**

READ, RESTORE, \_DATA

# RESUME Command

**Action:** error handling

**Syntax:** RESUME [NEXT] or  
RESUME [mar]  
*mar: name of a marker*

**Abbreviation:** resu

**Explanation:** The RESUME command is only meaningful with error capture (ON ERROR GOSUB proc) where it allows for a reaction to an error.

RESUME without a NEXT or mar jumps back to the command which caused the error.

RESUME NEXT jumps to the next command after the one that caused the error.

RESUME mar jumps, in case of an error, to the marker specified in mar. If, when an error occurs, variable FATAL = TRUE only RESUME mar can be used.

**Remarks:**

```
SCREEN 3
ON ERROR GOSUB err_handle
FOR i%=1 TO 5
    PRINT SQR(i%-3)
NEXT i%
//
PROCEDURE err_handle
    IF !FATAL
        PRINT "Square root of a negative number"
        RESUME NEXT
    ELSE
        PRINT "Invalid address. The program will be
            halted"
        RESUME hold
```



hold:  
EDIT  
ENDIF  
RETURN

**Remarks:** -

**See**

**Also:** ON ERROR, ON ERROR GOSUB

## RIGHT\$() Function

**Action:** returns the last *m* characters of a string expression.

**Syntax:** RIGHT\$(a\$,m%)  
*a\$*: *sexp*  
*m%*: *iexp*

**Abbreviation:** -

**Explanation:** RIGHT\$(a\$,m%) returns the last *m%* characters of the string expression *a\$*. If *m%* is greater than the number of characters contained in *a\$* (spaces and CHR\$(0) are characters too!), the complete *a\$* is returned. If *m%* is not given, the last character in *a\$* is returned.

**Example:**

```
PRINT RIGHT$("Hello GFA",3) // prints GFA
PRINT RIGHT$("Hello GFA",20) // prints Hallo GFA
PRINT RIGHT$("Hello GFA",-1) // prints A
PRINT RIGHT$("Hello GFA") // prints A
```

**Remarks:** -

**See**

**Also:** LEFT\$(), MID\$()

## RINSTR() String function

**Action:** searches for a substring in a string expression, optionally from a given position. If the substring is found, the position at which the substring begins within the string expression is returned. If the substring is not found a 0 is returned.

**Syntax:** RINSTR(a\$,b\$[,m%]) or  
RINSTR([m%],a\$,b\$)  
*a\$,b\$:* *sexp*  
*m%:* *iexp*

**Abbreviation:** -

**Explanation:** RINSTR(a\$,b\$,m%) works in principle like INSTR(a\$,b\$,m%), except that the search for the substring begins at the end of the string expression a\$.

**Example:**

```
PRINT RINSTR("Hello GFA","ll",2) // prints 0
PRINT RINSTR("Hello GFA","ll") // prints 3
PRINT RINSTR("Hello GFA","ll",5) // prints 3
```

**Remarks:** -

**See**

**Also:** INSTR()

### RMDIR Command

**Action:** deletes a directory.

**Syntax:** RMDIR a\$  
*a\$: sexp; directory name*

**Abbreviation:** rmdi

**Explanation:** RMDIR a\$ (remove directory) deletes the directory with the name a\$, assuming it does not contain any subdirectories.

**Example:** RMDIR "C:\TEST"  
  
// Deletes the directory TEST on drive C.

**Remarks:** -

**See**  
**Also:** MKDIR

## RND Function

**Action:** generates a random number between 0 (inclusive) and 1 (exclusive).

**Syntax:** RND[(*x*)]  
*x*: *aexp*

**Explanation:** The parameter *x* is optional and has no effect.

**Example:** PRINT RND  
  
// Prints a random number between 0 and 1.

**Remarks:** -

**See**

**Also:** RANDOM(), RAND(), RANDOMIZE

### ROL() Function

**Action:** rotates a bit pattern left.

**Syntax:** ROL(*m*,*n*) or  
ROL&(*m*,*n*) or  
ROL|(*m*,*n*)

*m*,*n*: *iexp*

**Explanation:** ROL(*m*,*n*) shifts the bit pattern of a 32-bit integer expressions *m*, *n* places left (ROL = ROTate Left) and "wraps around" the bits moved off the left end to the right end again. The resulting new value is, optionally, stored in a variable. ROL&(*m*,*n*) and ROL|(*m*,*n*) rotate the bit pattern of a 16-bit or an 8-bit integer expression *m* respectively, *n* places left.

**Example:**

```
PRINT BIN$(202,16)           // prints 0000000011001010
PRINT BIN$(ROL(202,4),16)    // prints 0000110010100000
l%=ROL(202,4)
PRINT l%                     // prints 3232
PRINT BIN$(202,16)           // prints 0000000011001010
PRINT BIN$(ROL&(202,4),16)    // prints 0000110010100000
l&=ROL&(202,4)
PRINT l&                     // prints 3232
//
PRINT BIN$(202,8)            // prints 11001010
PRINT BIN$(ROL|(202,4),16)    // prints 10101100
l|=ROL|(202,4)
PRINT l|                     // prints 172
```

**Remarks:** -

**See**

**Also:** SHL, SHR, ROR



### ROUND() Numeric function

**Action:** rounds off the numeric expression *x*.

**Syntax:** ROUND(*x*[,*n*])

*x*: *aexp*

*n*: *iexp*

**Explanation:** If *n* is equal to zero the rounding is performed to a whole number just like with ROUND(*x*). If *n* is positive, *x* is rounded off to *n* decimal places. If *n* is negative, the rounding is performed to the nearest integer.

**Example:**

```
PRINT ROUND(PI)           // prints    3
PRINT ROUND(PI,2)         // prints    3.14
PRINT ROUND(155,-1)       // prints    160
```

**Remarks:** -

**See**

**Also:** -



## RSET String command

**Action:** moves a string expression, right justified, to a string.

**Syntax:** RSET a\$=b\$  
a\$: svar  
b\$: sexp

**Abbreviation:** -

**Explanation:** RSET a\$=b\$ will, first of all, replace all characters in a\$ with spaces. Next, b\$ is moved into a\$ right justified. If b\$ contains more characters than a\$, then only as many characters as there are "places" for in a\$ are moved.

**Example:**

```
a$=STRING$(15,"-")
b$="Hello GFA
PRINT a$'LEN(a$) // prints----- 15

PRINT b$'LEN(b$) // printsHello GFA 9

RSET a$=b$
PRINT a$'LEN(a$) // prints      Hello GFA 15
```

**Remarks:** -

**See**

**Also:** LSET, MID\$

## RUBBERBOX Command

**Action:** cuts out a rectangular segment of the screen.

**Syntax:** RUBBERBOX *x1,y1,w1,h1*[*x2,y2,w2,h2*],*w3,h3*  
*x1,y1,w1,h1,x2,y2,w2,h2,w3,h3: iexp*

**Abbreviation:** rub *1,y1,w1,h1*[*x2,y2,w2,h2*],*w3,h3*

**Explanation:** RUBBERBOX can only be used in the graphic mode by pressing the left mouse button.

Given are the coordinates of the upper left corner as well as the minimal width and height. By moving the mouse the size of the rectangle can be changed (rubber band effect) as long as the left mouse button is held down. When the mouse button is released the width and height are returned.

By specifying the negative width and height the rectangle can be drawn in the upper left direction.

**Example:**

```
SCREEN 16           // EGA mode
REPEAT
  REPEAT
    MOUSE x%,y%,k%
  UNTIL k%
  RUBBERBOX x%,y%,-1000,-1000,a%,b%
  COLOR RAND(_C), RAND(_C)
  DEFFILL RAND(37)
  PBOX x%,y%,x%+a%,y%+b%
UNTIL MOUSEK AND 2
SCREEN 3
```

```
// This program enables drawing of rectangles in
// different colours with the mouse.
```

**Remarks:** -

**See**

**Also:** DRAGBOX

## RUN Command

**Action:** runs the program in memory.

**Syntax:** RUN [a\$]  
*a\$: sexp*

**Abbreviation:** -

**Explanation:** RUN runs the program in memory. If the optional string expression a\$ is specified the GFA-BASIC program is first loaded and then run. When the file extension is left out .GFA is used.

**Example:**

```
RUN // runs the current
// program.
RUN"C:\GFA\TEST.GFA" // loads the program TEST.GFA
// from subdirectory GFA in
// partition C and runs it.
```

**Remarks:** -

**See**  
**Also:** CHAIN, EXEC



### SAVE Command

**Action:** saves the current GFA-BASIC program.

**Syntax:** SAVE a\$  
a\$: *sexp*

**Abbreviation:** sa

**Explanation:** SAVE a\$ saves a BASIC program.

**Example:** SAVE"C:\TEST.GFA"

// Saves the current program under the name TEST.GFA  
// in partition C on the hard drive.

**Remarks:** -

**See**

**Also:** PSAVE, LIST

## SCALE() Function

**Action:** divides an integer variable into the product of two other integer variables.

**Syntax:** SCALE(i,j,k)  
*i,j,k: ivar*

**Explanation:** SCALE(i,j,k) multiplies the 16-bit integer numbers i and j, producing a 32-bit integer result. This result is then divided by the 16-bit integer number k and the 16-bit quotient is returned. This is somewhat faster than  $i*j/k$ .

**Example:**

```
i&=64
j&=8
k&=16
PRINT SCALE(i&,j&,k&) // prints 32
```

**Remarks:** One possible applications of this function is for scaling of graphics.

**See**

**Also:** -

## SCREEN Command

**Action:** sets the screen mode.

**Syntax:** SCREEN *n*  
*n*: *icxp*

**Abbreviation:** SCR *n*

**Explanation:** SCREEN *n* sets the screen mode depending on the value in *n*. The screen modes are as follows:

<i>n</i> =type	Number of colours	Text format	Buffer address
0	Text 16	40 * 25	B8000
1	Text 16	40 * 25	B8000
2	Text 16	80 * 25	B8000
3	Text 16	80 * 25	B8000
4	Graphic 4	320 * 200	B8000
5	Graphic 4	320 * 200	B8000
6	Graphic 2	640 * 200	B8000
7	Monochrome text	80 * 25	B0000
8	Graphic 16	160 * 200	B0000
9	Graphic 16	40 * 25	B0000
10	Graphic 4	80 * 25	B0000
11	RESERVED		
12	RESERVED		
13	Graphic 16	320 * 200	A0000
14	Graphic 16	640 * 200	A0000
15	Monochrome graphic	640 * 350	A0000
16	Graphic 16	640 * 350	A0000
17	Graphic 2	640 * 480	A0000
18	Graphic 16	640 * 480	A0000
19	Graphic 256	320 * 200	A0000



**Example:** -

**Remarks:** SCREEN n is equivalent to the system call INT 10H, AH=00H.

After the SCREEN activates the selected graphic mode

\_X returns the horizontal and

\_Y the vertical screen resolution (as long no window is opened, otherwise \_X and \_Y contain the inside width and height of the window respectively),

\_C the number of possible colours,

\_TS the segment address of the text screen,

\_MD currently active graphic mode; which is equivalent to the value specified with SCREEN,

\_ADAPT the graphic adapter type:

0 = MDA            3 = EGA

1 = HGC           4 = VGA

2 = CGA           5 = as a rule, it doesn't go beyond VGA

**See**

**Also:** -

## SCROLL ON and SCROLL OFF Commands

**Action:** turns scrolling on and off

**Syntax:** SCROLL OFF/ON

**Abbreviation:** sc off/on

**Explanation:** SCROLL OFF disables scrolling when printing on the last line, so that the last line is constantly overwritten.  
SCROLL ON turns the scrolling back on.

**Example:**

```
SCREEN 3
LOCATE 1,24
FOR i%=1 TO 10
    PRINT "Test line Test line Test line"i%
NEXT i%
SCROLL OFF
LOCATE 1,24
REPEAT
    UNTIL LEN(INKEY$)
FOR i%=1 TO 10
    PRINT "Test line Test line Test line"i%
NEXT i%
REPEAT
    UNTIL LEN(INKEY$)
```

**Remarks:** -

**See**

**Also:** WRAP ON/OFF

## SEEK # Command

**Action:** absolute positioning of the data pointer

**Syntax:** SEEK #*n*,*pos*  
*n*: *iexp*; *channel number*  
*pos*: *iexp*; *position*

**Abbreviation:** see #*m*,*pos*

**Explanation:** SEEK #*n*,*pos* enables access to index sequential files. *n* specifies the channel number from (0 to 99) previously opened with OPEN. SEEK can only be used with files and not with peripheral devices. When *pos* is specified it causes the data pointer to be positioned to the byte given in *pos*. *pos* can, therefore, only assume values between 0 and total file length. For *pos*=0 the data pointer is set to start of file.

The negative *pos* leads to positioning from the end of the file.

**Example:**

```
OPEN "o",#1,"A:\TEST.DAT"
PRINT #1,STRING$(20,"*")
SEEK #1,10
OUT #1,ASC("!"),ASC("0")
CLOSE #1
OPEN "i",#1,"A:\TEST.DAT"
LINE INPUT #1,a$
CLOSE #1
PRINT a$           // prints
//                *****!0*****
```

**Remarks:** -

**See**

**Also:** RELSEEK

### SELECT...CASE Structure

**Action:** A conditional command which enables execution of specified program segments depending on an integer expression.

**Syntax:**

```
SELECT x
CASE value1[,value2,...]
    programsegment]
[CONT]
[CASE value1[,value2,...]]
    programsegment]
[CONT]
[CASE value1[,value2,...]]
    programsegment]
.
.
[CONT]
[DEFAULT]
    programsegment]
ENDSELECT
```

**x:** *iexp or a string - only the first four characters of which are significant.*

**value1,value2,...:** *an integer or string constant of up to four characters, % variable, & variable, | variable. Also TO value, value TO or value1 TO value2*

**Explanation:**

SELECT takes one of the CASE conditional branches depending on the value of "x". The process begins by selecting and evaluating the first CASE conditional branch, to test if "x" corresponds to at least one of the values after CASE. If it does, the program segment following this CASE is executed and a branch is taken to the program line following the ENDSELECT.

If "x" does not correspond to any values in the first CASE conditional branch the next CASE is selected.

The program segment in a CASE can be terminated with a CONT. This will cause the program control to pass to the next CASE or DEFAULT instead of a branch to the program line after the ENDSELECT.

Every CASE must be followed by at least one value. When entering a list of values its elements must be separated by commas. Furthermore, GFA-BASIC will also accept a range of values. CASE TO value corresponds to a range of whole numbers whose elements are less than or equal to value. CASE value TO corresponds to the range of whole numbers whose elements are greater than or equal to value. CASE value1 TO value2 corresponds to the range of whole numbers whose elements are greater than or equal to value1 and less than or equal to value2.

If no CASE conditional statement is satisfied the program segment after the optional DEFAULT is executed and a branch is taken to the program line following the ENDSELECT; if there is no DEFAULT, a branch to the program line following the ENDSELECT is taken immediately.

## Commands and functions

---

The `SELECT...CASE` conditional statement can therefore assume the following structures:

```
= value: CASE value
<= value: CASE TO value
=> value: CASE value TO and
(>= value1) AND (<= value2): CASE value1 TO value2
```

### Example:

```
REPEAT
  a%=ASC(INKEY$)
  SELECT a%
  CASE 65 TO 90
    PRINT "Capital letter"
  CASE 97 TO 122
    PRINT "Lower case letter"
  DEFAULT
    PRINT "Not a letter or umlaut"
  ENDSELECT
UNTIL a%=27
```

```
// A character entered from the keyboard is evaluated
// in this loop using the SELECT...ENDSELECT struc-
// ture. If the ASCII code of the character is bet-
// ween 65 and 90 inclusive, 'Capital letter' is
// printed out. If the ASCII code falls between 97
// and 122, 'Lower case letter' is displayed. If
// neither of the two conditions is satisfied, 'Not a
// letter or umlaut' is shown.
// The loop is terminated by entering a character
// with ASCII code 27 (= Esc).
```

```
n%=3
m%=2
SELECT n%*m%
CASE TO 10
  PRINT "n*m <= 10"
CONT
```

```
CASE TO 5
  PRINT "n*m <= 5"
CASE 5 TO 9
  PRINT "5 < n*m < 10"
DEFAULT
  PRINT "n*m > 10"
ENDSELECT
```

```
// Gives 'n*m <= 10' and 'n*m <= 5'. The output of
// 'n*m <= 5' is forced by CONT after the first
// CASE.
```

```
a%=ASC(INKEY$)
SELECT a%
CASE 65,66,67,85 TO 90
  PRINT "A, B, C, D, U, V, W, X, Y or Z"
CASE TO 64
  PRINT "Not a letter"
CASE 123 TO
  PRINT "Not a letter or umlaut"
CASE 97 TO 122
  PRINT "Lower case letter"
ENDSELECT
```

**Remarks:**

SWITCH can be used instead of SELECT, OTHERWISE or CASE ELSE instead of DEFAULT and ENDSWITCH instead of ENDSELECT.

The common structures can be evaluated using IF...ENDIF conditional statement. The SELECT...CASE if often considerably faster than IF...ELSEIF.

**See****Also:**

IF...ENDIF, ON...GOSUB

# SETTIME Command

**Action:** sets the system time and date.

**Syntax:** SETTIME = time\$,date\$  
*time\$, date\$: sexp*

**Explanation:** SETTIME = time\$,date\$ sets the system time in the following format:

HH:MM:SS (Hours:Minute:Seconds)

and the system date in the following format:

DD.MM.YYYY (Day.Month.Year) or

MM.DD.YYYY (Month.Day.Year; US format)

This format is set with the MODE command.

**Example:** SETTIME="23:50:39", "01.10.1990"

**Remarks:** -

**See**

**Also:** DATE\$ =, TIME\$ =



## SGN() Numeric function

**Action:** returns the sign of a numeric expression.

**Syntax:** SGN(*x*)  
*x*: *aexp*

**Explanation:** -

**Example:**

```
PRINT SGN(-210)      // prints  -1
PRINT SGN(ABS(5-10)) // prints   1
PRINT SGN(0)         // prints   0
```

**Remarks:** The value returned by the SGN() function depends on the sign of the argument *x*:

*x* < 0 returns -1,  
*x* = 0 returns 0 and  
*x* > 0 returns 1.

**See**

**Also:** ABS()

## SHELL Command

**Action:** runs COMMAND.COM.

**Syntax:** SHELL *t*  
*t:* *sexp*

**Abbreviation:** sh

**Explanation:** SHELL runs the MS-DOS command interpreter and so enables execution of DOS commands from within a GFA-BASIC program.

**Example:** SHELL "CHKDSK a: /f" // tests the disk in  
// drive A:

**Remarks:** -

**See**

**Also:** EXEC(), EXEC

## SHL() Function

**Action:** shifts a bit pattern left.

**Syntax:** SHL(*m*,*n*) or  
SHL&(*m*,*n*) or  
SHL|(*m*,*n*)  
*m*,*n*: *exp*

**Explanation:** SHL(*m*,*n*) shifts the bit pattern of a 32-bit integer expressions *m*, *n* places left (SHL = SHift Left) and, optionally, stores the new value in a variable. SHL&(*m*,*n*) and SHL|(*m*,*n*) shift the bit pattern of a 16-bit or an 8-bit integer expression *m* respectively, *n* places left.

**Example:**

```
PRINT BIN$(202,16)           // prints 0000000011001010
PRINT BIN$(SHL(202,4),16) // prints 0000110010100000
1%=SHL(202,4)
PRINT 1%                      // prints 3232
//
PRINT BIN$(202,16)           // prints 0000000011001010
PRINT BIN$(SHL&(202,4),16)// prints 0000110010100000
1&=SHL&(202,4)
PRINT 1&                     // prints 3232
//
PRINT BIN$(202,8)            // prints 11001010
PRINT BIN$(SHL|(202,4),8) // prints 10100000
1|=SHL|(202,4)
PRINT 1|                      // prints 160
```

**Remarks:** *m* < < *n* is synonymous with SHL(*m*,*n*) and can be used instead.

As long as the result of the shift does not exceed the given width, **SHL(m,n)** is equivalent to a multiplication of  $m$  with  $2^n$ .

**See**

**Also:**

**SHR, ROL, ROR, <<, >>**



## Commands and functions

---

**Remarks:** WORD() is synonymous with SHORT() and can be used instead.

**See**

**Also:** BYTE(), WORD(), CARD(), USHORT()

## SHORT{} Function

**Action:** reads a word (16 bits) from an address.

**Syntax:** SHORT{addr}  
*addr: address*

**Abbreviation:** -

**Explanation:** Reads a word (16 bits) from an address.

**Example:** -

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

WORD{} and INT{} are synonymous with SHORT{} and can be used instead.

addr: see the {} function.

**See**

**Also:** DPEEK(), BYTE{}, WORD{}, CARD{}, INT{}, LONG{}, {}, SINGLE{}, DOUBLE{}, USHORT{}, UWORD{}

## SHORT{} = Command

**Action:** writes a word (16 bits) to an address.

**Syntax:** SHORT{addr} = m  
*addr: address*  
*m: iexp*

**Abbreviation:** -

**Explanation:** Writes a word (16 bits) to an address.

**Example:** -

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

WORD{} = and INT{} = are synonymous with SHORT{} = and can be used instead..

addr: see the {} function.

**See**

**Also:** DPOKE(), BYTE{} =, WORD{} =, CARD{} =, INT{} =, LONG{} =, {} =, SINGLE{} =, DOUBLE{} =, USHORT{} =, UWORD{} =



## SHR() Function

**Action:** shifts a bit patter right.

**Syntax:** SHR(*m*,*n*) or  
SHR&(*m*,*n*) or  
SHR|(*m*,*n*)  
*m*,*n*: *iexp*

**Explanation:** SHR(*m*,*n*) shifts the bit pattern of a 32-bit integer expressions *m*, *n* places right (SHR = SHift Right) and, optionally, stores the new value in a variable. SHR&(*m*,*n*) and SHR|(*m*,*n*) shift the bit pattern of a 16-bit or an 8-bit integer expression *m* respectively, *n* places right.

**Example:**

```
PRINT BIN$(202,16)           // prints 0000000011001010
PRINT BIN$(SHR(202,4),16)    // prints 0000000000001100
1%=SHR(202,4)
PRINT 1%                     // prints 12
//
PRINT BIN$(202,16)           // prints 0000000011001010
PRINT BIN$(SHR&(202,4),16)   // prints 0000000000001100
1&=SHR&(202,4)
PRINT 1&                     // prints 12
//
PRINT BIN$(202,8)            // prints 11001010
PRINT BIN$(SHR|(202,4),8)    // prints 00001100
1|=SHR(202,4)
PRINT 1|                     // prints 12
```

**Remarks:**  $m > n$  is synonymous with SHR(*m*,*n*) and can be used instead.

As long as the result of the shift does not exceed the given width, SHR(*m*,*n*) is equivalent to a division of *m* by  $2^n$ .

**See**

**Also:**

SHL, ROL, ROR, < <, > >

## SIN() Function

**Action:** returns the sine of a numeric expression.

**Syntax:** SIN(*x*)  
*x*: *an expression; angle in radians*

**Explanation:** The sine of an angle in a right-angled triangle corresponds to a quotient between the hypotenuse and the side opposite the angle.

The value of *x* is given in radians.

**Example:**

```
PRINT SIN(0)           // prints    0
PRINT SIN(PI/2)        // prints    1
PRINT SIN(PI)          // prints    0
PRINT SIN(3*PI/2)      // prints   -1
PRINT SIN(2*PI)        // prints    0
```

**Remarks:** SIN() is the reverse function of ASIN().

**See**

**Also:** SINC(), COS(), COSQ(), TAN(), ASIN(), ACOS(), ATN(), ATAN()

### SINGLE{} Function

**Action:** reads a 32-bit number in IEEE double format from an address.

**Syntax:** SINGLE{addr}  
*addr: address*

**Explanation:** same as for DOUBLE{}

**Example:** -

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

addr: see the {} function.

**See**

**Also:** BYTE{}, CARD{}, WORD{}, INT{}, LONG{}, {},  
DOUBLE{}

## SINGLE{} Command

**Action:** writes a 32-bit number in IEEE double format to an address.

**Syntax:** SINGLE{addr} = x  
*addr: address*

**Explanation:** SINGLE{} is also equivalent to DOUBLE{}.

**Example:**

```
a=1.2345
a%=0
PRINT HEX$({*a},8)''HEX$({*a+4},8)
SINGLE{*a%}=a
PRINT HEX$(a%,8)

// prints
// 126E978D 3FF3C083 and 3F9E0419.
```

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

SINGLE{} converts floating point numbers, which are internally represented with high accuracy (64 bits), to floating point numbers using only 32 bits at a cost of lower accuracy (8 places only).

addr: set the {} function.

**See**

**Also:** BYTE{}=, CARD{}=, WORD{}=, INT{}=,  
LONG{}=, {}=, DOUBLE{}=

### SINH() Function

**Action:** returns the hyperbolic sine of a numeric expression.

**Syntax:**  $\text{SINH}(x)$   
*x:* *aexp*

**Explanation:** The hyperbolic sine is defined with the function:  
$$\text{SINH}(x) = (\text{EXP}(x) - \text{EXP}(-x)) / 2$$

**Example:**  

```
PRINT SINH(2.14)           // prints 4.19089...  
PRINT SINH(ARSINH(2.14))  // prints 2.14
```

**Remarks:**  $\text{SINH}()$  is the reverse function of  $\text{ARSINH}()$ .

**See**

**Also:**  $\text{COSH}()$ ,  $\text{TANH}()$ ,  $\text{ARSINH}()$ ,  $\text{ARCOSH}()$ ,  
 $\text{ARTANH}()$

## SINQ() Function

**Action:** returns the extrapolated sine of a numeric expression.

**Syntax:** SINQ(x)  
*x:* aexp; angle in degrees

**Explanation:** For SINQ() GFA-BASIC uses an internal table with sine values in one degree steps. SINQ(x) expects, therefore, the expression x to be in degrees. The intermediate values of function x are extrapolated in 1/16- degree steps. This accuracy is sufficient for plotting of graphs on the screen, particularly when there is no co-processor, since this function is several times faster than SIN(x).

**Example:**

```
PRINT SINQ(180)      // prints    0
PRINT SIN(PI)        // prints    0
```

**Remarks:** -

**See**

**Also:** SIN(), COS(), COSQ(), TAN(), ASIN(), ACOS(),  
ATN(), ATAN()

### SIZEW # Command

**Action:** changes the size of a window.

**Syntax:** SIZEW #n,w,h  
*n,w,h: iexp*

**Abbreviation:** -

**Explanation:** SIZEW #n,w,h changes the size of the window specified in n (1, 2, 3 or 4). w specifies the new width and h the new height. The position of the window (upper left corner) remains unchanged.

**Example:**

```
SCREEN 16                // EGA mode
OPENW #1,10,10,200,100,-1
REPEAT
UNTIL LEN(INKEY$)
SIZEW #1,400,200
REPEAT
UNTIL LEN(INKEY$)
CLOSEW #1
EDIT
```

```
// Draws a window on the screen and waits for a
// keypress. The window size is then changed.
```

**Remarks:** -

**See**

**Also:** OPENW, CLOSEW, TITLEW, INFOW, FULLW,  
CLEARW, TOPW, MOVEW



## SPACE\$() String function

**Action:** creates a string consisting of a specified number of spaces.

**Syntax:** SPACE\$(m%)  
*m%: iexp*

**Abbreviation:** -

**Explanation:** -

**Example:**

```
b$="Hello"  
c$="GFA"  
a$=SPACE$(5)  
PRINT b$+a$+c$           // prints Hello      GFA
```

**Remarks:** -

**See**

**Also:** STRING\$()

### SPC Print Option

**Action:** prints spaces.

**Syntax:** SPC(*m*)  
*m*: *iexp*

**Abbreviation:** -

**Explanation:** When used with PRINT, SPC prints the specified number of spaces.

**Example:**

```
a$="Hello"  
b$="GFA"  
PRINT a$;SPC(5);b$;SPC(3);42
```

```
// prints  
// Hello      GFA      42
```

**Remarks:** -

**See**

**Also:** TAB

## SQR() Numeric function

**Action:** returns the positive square root of a numeric expression.

**Syntax:** SQR(x)  
*x:* *aexp*

**Explanation:**

**Example:** PRINT SQR(PI\*5.3+1) // prints 4.20124...

**Remarks:** If the function argument x is less than 0, SQR(x) reports an error.

**See**

**Also:** Exponential operator (^)

## SSORT Function

**Action:** sorts the elements in an array by its size using the Shell-Metzner method.

**Syntax:**

```
SSORT x(s)[,n][,m%()]
SSORT x$(s) [WITH n()][OFFSET o][[,n[,m%()]]]
s:          + or - for ascending or descending
              order
n,o:        iexp
x():        one dimensional array
x$():       one dimensional string array
n():        one dimensional integer array with 1,2 or 4
              byte variables
m%( )       one dimensional integer array with 4 byte
              variables
```

**Abbreviation:** -

**Explanation:** The s enclosed in round brackets can be replaced with a "+", a "-" or it may be left out. "+" or no specification results in arrays y() and m%( ) being sorted in ascending order. In this case, after the sorting, the smallest array element assumes the smallest index (0 for OPTION BASE 0 or 1 for OPTION BASE 1). "-" results in the array being sorted in descending order. In this case, after the sorting, the biggest array element assumes the smallest index.

The parameter n specifies that only the first n elements of the array should be sorted. For OPTION BASE 0 these are the elements with indices 0 to "n"-1, and for OPTION BASE 1 the elements with indices 1 to "n". If n is given explicitly, it can be followed by a LONG integer array, which will be sorted together with the array a(), that is to say, each swap in array a() will also be performed in array m%( ). This is particu-

larly useful when the sorted array `a()` contains the sort key (for example the postal code), while other arrays contain the additional information whose information must maintain the same order as the keys.

When sorting string arrays (`x$()`) a sort criterion can be specified with `WITH`, in form of an array with at least 256 elements. If `WITH` is not given the normal ASCII table is used as the sort criterion.

Optionally, `OFFSET o` can specify the number of characters at the start of the string to ignore during sorting.

**Example:**

```
OPTION BASE 1
a:
DATA 10,-3,5,21
c:
DATA "A","B","C","D"
d:
DATA "Who","How","What","Where"
DIM a(4),b%(4),c$(4),d$(4)
RESTORE a
MAT READ a()
RESTORE c
FOR i%=1 TO 4
  READ c$(i%)
  b%(i%)=i%
NEXT i%
RESTORE d
FOR i%=1 TO 4
  READ d$(i%)
NEXT i%
FOR i%=1 TO 4
  PRINT STR$(a(i%),5,2)''
  PRINT STR$(b%(i%),5,2)''
  PRINT c$(i%)''
  PRINT d$(i%)
```

## Commands and functions

---

```
NEXT i%
PRINT
SSORT a(),4,b%()
FOR i%=1 TO 4
    PRINT STR$(a(i%),5,2)''
    PRINT STR$(b%(i%),5,2)''
    PRINT c$(b%(i%))''
    PRINT d$(b%(i%))
NEXT i%

// prints first of all (unsorted)
// 10.00 1.00 A Who
// 3.00 2.00 B How
// 5.00 3.00 C What
// 20.00 4.00 D Where
// and then (sorted)
// 3.00 2.00 B How
// 5.00 3.00 C What
// 10.00 1.00 A Who
// 20.00 4.00 D Where
```

**Remarks:**

-

**See**

**Also:**

QSORT

## STOP Command

**Action:** halts a GFA-BASIC program.

**Syntax:** STOP

**Abbreviation:** -

**Explanation:** The STOP command halts the program on the line with the STOP command.

After a STOP the variables and arrays can be printed and inspected. The continuation of the program after a STOP is performed by entering CONT.

**Example:**

```
SCREEN 16           // EGA mode
OPENW #1,10,10,100,100,-1
FULLW #1
FOR i%=1 TO 100
  IF MOD(i%,10)=0
    PRINT i%
    STOP
  ENDIF
NEXT i%
~INTR(33,_AH=8)
EDIT

// Performs a STOP whenever the counter is a multiple
// of 10.
```

**Remarks:** -

**See Also:** CONT

### STORE # Command

**Action:** fast save of a string array as a text file

**Syntax:** STORE #n,a\$()[,m]  
*n:* iexp; channel number  
*a\$():* one dimensional string array  
*m:* iexp

**Abbreviation:** -

**Explanation:** STORE saves the complete string array through the opened channel *n* (from 0 to 99) to a file. The individual strings in the file saved with STORE are separated by a CR/LF. The parameter *m* is optional and defines how many strings from *a\$()* should be written to the text file.

**Example:**

```
DIM a$(1000)
FOR i%=0 TO 499
    a$(i%)="Hello GFA"
NEXT i%
OPEN "o",#1,"A:\TEST.TXT"
STORE #1,a$(),500
CLOSE #1
```

```
// The Hello GFA string is first written 500 times to
// the a$() array.
// After that the first 500 strings in a$() are saved
// to the file TEST.TXT on drive A using the STORE
// command.
```

**Remarks:** STORE is for strings what BSAVE or BPUT are for general arrays and memory areas.

**See**  
**Also:** RECALL



## STR\$( ) Function

**Action:** converts a numeric expression into a string.

**Syntax:** STR\$(x[,m,n])  
*x:* aexp  
*m,n:* iexp

**Explanation:** STR\$(x,m) converts x into a string of m length. If m is greater than the number of characters needed to represent x, the string is padded with leading spaces. If m is smaller than the number of characters needed to represent x, the string is truncated from the right.

STR\$(x,m,n) converts x into a string of m length with n decimal places. The last decimal place is rounded off. Out of the total length m, n + 1 places are reserved (n places for the decimal part and one place for the decimal point).

**Example:**

```
PRINT STR$(3*4+2)           // prints  14
a$=STR$(3*4+2)
PRINT a$                     // prints  14
PRINT STR$(123.456,7)       // prints 123.456
PRINT STR$(123.456,9)       // prints  123.456
PRINT STR$(123.456,5)       // prints 123.4
PRINT STR$(123.456,7,3)     // prints 123.456
PRINT STR$(123.456,7,5)     // prints  3.45600
PRINT STR$(123.456,7,2)     // prints 123.46
PRINT STR$(123.456,9,3)     // prints  123.456
```

**Remarks:** -

**See**

**Also:** PRINT USING, VAL(), VAL?()

## STRING\$( ) Function

**Action:** creates a string consisting of a string expression repeated specified number of times.

**Syntax:**           STRING\$(m,a\$)   or  
                  STRING\$(m,n)  
                  *a\$:*           *sexp*  
                  *n,m:* *iexp*

**Abbreviation:**    -

**Explanation:**    STRING\$(5,"Hello") creates a string in which the string expression "Hello" is repeated 5 times one after the other. If an integer expression is used instead of a string expression, STRING\$(m,n) creates a string in which CHR\$(n) repeats m% times.

**Example:**           PRINT STRING\$(3,"GFA") // prints    GFAGFAGFA  
                  a\$=STRING\$(3,64)  
                  PRINT a\$                       // prints    AAA

**Remarks:**        -

**See**

**Also:**            SPACE\$( )

## SUB() Function

**Action:** subtracts two integer expressions.

**Syntax:** SUB(*i,j*)  
*i,j*: *iexp*

**Explanation:** SUB(*i,j*) calculates the difference between integer expressions *i* and *j* and, optionally stores it in a variable.

**Example:**

```
PRINT SUB(5^3,4*20+3) // prints 42
1%=SUB(5^3,4*20+3)
PRINT 1%              // prints 42
```

**Remarks:** The ADD(), SUB(), MUL() and DIV() functions can be mixed freely with each other. For example:

```
1%=SUB(5^3,4*20+3) or
1%=SUB(5^3,ADD(MUL(4,20),3))
```

**See**

**Also:** ADD(), MUL(), DIV(), MOD(), PRED(), SUCC()

### SUB Command

**Action:** subtracts a numeric expression from a numeric variable.

**Syntax:** SUB x,y  
x: *avar*  
y: *aexp*

**Abbreviation:** -

**Explanation:** SUB x,y subtracts the expression y from value in variable x.

**Example:**

```
x=57
SUB x,3*5
PRINT x           // prints 42
```

**Remarks:** Although SUB can be used with any numeric variable, the usage of integer variables is recommended in order to achieve the maximum optimisation for speed.

Instead of SUB x,y

```
x=x-y
x:=x-y or
x-=y
```

can be used also.

When integer variables are used SUB doesn't test for overflow!

**See**

**Also:** DEC, INC, ADD, MUL, DIV, ++, --, +=, -=, \*=, /=

## SUCC() Function

**Action:** calculates the first natural number greater than an integer expression.

**Syntax:** SUCC(*n*)  
*n*: *iexp*

**Explanation:** SUCC(*n*) returns the first natural number greater than the integer expression *n*.

**Example:**

```
PRINT SUCC(4*10+1)    // prints    42
1%=SUCC(4*10+1)
PRINT 1%              // prints    42
```

**Remarks:** -

**See**

**Also:** ADD(), SUB(), MUL(), DIV(), MOD(), PRED()

## SUCC() String function

**Action:** returns a character whose ASCII value is one greater than the first character of a string expression.

**Syntax:** SUCC(a\$)  
*a\$: sexp*

**Abbreviation:** -

**Explanation:** -

**Example:** PRINT SUCC("Hello world") // prints I

**Remarks:** SUCC(a\$) corresponds to CHR\$(SUCC(ASC(a\$))).

**See**

**Also:** PRED()

## SWAP() Function

**Action:** swaps the low and high words.

**Syntax:** SWAP(*n*)  
*n*: *iexp*

**Explanation:** The SWAP(*n*) function swaps both halves of a long-word. The most common usage is to separate the segment and offset portions of an address or to convert files from Intel (80x86) to Motorola format (680x0).

**Example:** ~INTR(\$21,\_AH=\$2F)  
dta%=SWAP(\_ES)|\_BX  
  
// Simulates the function dta%=FGETDTA().

**Remarks:** -

**See**

**Also:** ROL, SHR,...

### SWAP Command

**Action:** swaps two variables of the same type.

**Syntax:** SWAP x,y  
*x,y: variables of any type*

**Abbreviation:** -

**Explanation:** Beside simple swapping of two variables, SWAP can also be used to swap two array descriptors.

**Example:**

```
x=2
y=5
SWAP x,y
PRINT x,y           // prints    5    2
DIM a(3),b(5)
MAT BASE 1
a:
DATA 5,3.14,19
b:
DATA 19,29,21,22,23
RESTORE a
MAT READ a()
RESTORE b
MAT READ b()
SWAP a(),b()
MAT PRINT b()       // prints    5,3.14,19
```

**Remarks:** -

**See**

**Also:** -



## SWITCH Condition test

**Action:** a part of the SELECT...CASE structure. It contains the condition test.

**Syntax:** SWITCH condition

**Explanation:** see "SELECT...CASE Structure"

**Example:** -

**Remarks:** CASE is synonymous with SWITCH and can be used instead.

**See  
Also:** -

### SYSCOL Command

**Action:** sets colours for the menu bar, pull-down menus, pop-up menus, window frames and 3D effects

**Syntax:** SYSCOL o,vc,hc  
o,vc,hc: *iexp*

**Explanation:** SYSCOL o,vc,hv sets the colours for menu bar, pull-down menus, pop-up menus, window frames and 3D effects. o indicates the object as follows

o = 0	menu bar
o = 1	pull-down menu
o = 2	pop-up menu
o = 3	window frame
o = 4	3D effects
o = 5	in windows and pop-up menus
o = 6	desktop
o = 7	desktop fill pattern (similar to DEFFILL)
o = 8	alerts and file selector

vc is the character colour and hc the background colour.

#### For CGA resolution:

##### 1. Palette

vc or hc = 1 cyan  
vc or hc = 2 violet  
vc or hc = 3 white

##### 2. Palette

vc or hc = 1 green  
vc or hc = 2 red  
vc or hc = 3 yellow

### For EGA or VGA resolution:

vc or hc = 0	( 0)	black
vc or hc = 1	( 1)	blue
vc or hc = 2	( 2)	green
vc or hc = 3	( 3)	cyan
vc or hc = 4	( 4)	red
vc or hc = 5	( 5)	magenta
vc or hc = 6	(20)	brown
vc or hc = 7	( 7)	light grey
vc or hc = 8	(56)	dark grey
vc or hc = 9	(57)	light blue
vc or hc = 10	(58)	light green
vc or hc = 11	(59)	light cyan
vc or hc = 12	(60)	light red
vc or hc = 13	(61)	light purple
vc or hc = 14	(62)	light yellow
vc or hc = 15	(63)	white

### Example:

```
SCREEN 16
SYSCOL 3,1,4
OPENW #1,10,10,400,200,-1
REPEAT
UNTIL LEN(INKEY$)
SCREEN 3
```

```
// Opens a window and sets the window frame to red
// with blue characters.
```

### Remarks:

-

### See

### Also:

COLOR

## SYSTEM Command

**Action:** terminates a GFA-BASIC program and returns back to the calling program.

**Syntax:** SYSTEM [*i*]  
*i*: *iexp*

**Abbreviation:** -

**Explanation:** The SYSTEM command terminates the current GFA-BASIC program and returns to the calling program. Optionally, a 8 bit integer value can be returned to the calling program. The following convention applies:

*i* = 0 the program was executed without error.

*i* > 0 an internal program error has occurred.

*i* < 0 an operating system error has occurred.

**Example:**

```
SCREEN 16           // EGA mode
OPENW #1,10,10,200,100,-1
REPEAT
  a$=inkey$
UNTIL a$=chr$(27)
CLOSEW #1
SCREEN 3
SYSTEM

// Draws a window on the screen, waits for the
// pressing of the Esc key, turns the text mode on
// and returns to the calling program (operating
// system).
```

**Remarks:** QUIT is synonymous with SYSTEM and can be used instead.

**See**

**Also:** QUIT, EDIT

### TAB() Print option

**Action:** prints spaces.

**Syntax:** TAB(*n*)  
*n*: *iexp*

**Abbreviation:** -

**Explanation:** Prints spaces until cursor column is equal to *n*. If *n* is smaller than the current cursor column a line break is performed (CR + LF).

**Example:** PRINT "Hello";TAB(10);"GFA"

// prints Hello      GFA

**Remarks:** -

**See**

**Also:** LOCATE, SPC, VTAB, HTAB

## TAN() Function

**Action:** returns the tangent of a numeric expression.

**Syntax:** TAN(*x*)  
*x*: *anexp*; angle in radians

**Explanation:** The tangent of an angle corresponds to the quotient of two short sides in a right-angled triangle. The value of *x* is given in radians.

**Example:**

```
PRINT TAN(PI)           // prints    0
PRINT TAN(PI/4)         // prints    1
```

**Remarks:** TAN() is the reverse function of ATN() or ATAN().

**See**

**Also:** SIN(), SINQ(), COS(), COSQ(), ASIN(), ACOS(),  
ATN(), ATAN()

### TANH() Function

**Action:** returns the hyperbolic tangent of a numeric expression.

**Syntax:** TANH(*x*)  
*x*: *aexp*

**Explanation:** The hyperbolic tangent is defined as the function:

$$\begin{aligned}\text{TANH}(x) &= (\text{EXP}(x) - \text{EXP}(-x)) / (\text{EXP}(x) + \text{EXP}(-x)) \\ &= (1 - \text{EXP}(-2*x)) / (1 + \text{EXP}(-2*x))\end{aligned}$$

The function  $y = \text{TANH}(x)$  returns values between -1 and +1.

**Example:**

```
PRINT TANH(2.14)           // prints 0.97269...
PRINT TANH(ARTANH(-0.5))  // prints -0.5...
```

**Remarks:** TANH() is the reverse function of ARTANH().  
The hyperbolic cotangent area is obtained with:  
 $\text{COTH}(x) = 1/\text{TANH}(x)$  or  $\text{COTH}(x) = \text{COSH}(x)/\text{SINH}(x)$

**See**

**Also:** SINH(), COSH(), ARSINH(), ARCOSH(),  
ARTANH()



## TBOX Command

**Action:** draws a rectangle in text mode using the graphic characters from the IBM character set.

**Syntax:** TBOX *n,x,y,w,h*  
*n,x,y,w,h:* *ixep*

**Abbreviation:** tb *n,x,y,w,h*

**Explanation:** Using the special characters from the IBM character set, TBOX *n,x,y,w,h* draws a frame whose upper left corner is specified in *x* and *y*, then width in *w* and height in *h*.

The frame style is defined with *n* as follows:

*n* = 0 build a frame out of space characters

*n* = 1 build a frame out of single lines

*n* = 2 build a frame out of double lines

**Example:**

```
SCREEN 3
TBOX 2,2,2,51,20
TCLIP 3,3 TO 50,19
a$="First, it is slightly cheaper; "
b$="and second it has the words "
c$="DON'T PANIC inscribed in large      "
d$="friendly letters on its cover"
PRINT
PRINT
PRINT a$;b$;c$;d$
REPEAT
UNTIL LEN(INKEY$)
```

```
// Writes a quote from D. ADAMS', The Hitchhiker's
// Guide to the Galaxy, Pocket Books, N.Y, 1981 to
// the screen.
```

## Commands and functions

---

**Remarks:**

-

**See**

**Also:**

TCLIP, TCLIP OFF, TPBOX, TPUT, TGET,  
TCOLOR

## TCLIP Command

**Action:** limits the output area in text mode.

**Syntax:** TCLIP x1,y1,x2,y2  
*x1,y1,x2,y2: iexp*

**Abbreviation:** tcl x1,y1,x2,y2

**Explanation:** TCLIP x1,y1,x2,y2 limits the output area in text mode. All output is then limited to the area defined by x1,y1 and x2,y2. x1 and y1 define the upper left corner and x2 and y2 the lower right corner of the clipping area.

This is not real clipping, since the lines which are too long are only broken up by TCLIP (provided that WRAP ON is active) and the output below the bottom edge causes the area to scroll up (provided that SCROLL ON is active).

**Example:**

```
SCREEN 3
PRINT STRING$("-",60)
TCLIP 0,0,50,30
PRINT STRING$("-",60)
TCLIP OFF
PRINT STRING$("-",60)
REPEAT
UNTIL LEN(INKEY$)
```

**Remarks:** TCLIP OFF reverts back to the full screen area.

**See**

**Also:** TCLIP OFF, TGET, TPUT, TBOX, TPBOX, TCOLOR

### TCLIP OFF Command

**Action:** removes the output area clipping in text mode.

**Syntax:** TCLIP OFF

**Abbreviation:** tcl off

**Explanation:** TCLIP x1,y1,x2,y2 limits the output area in text mode. All output is then limited to the area defined by x1,y1 and x2,y2. x1 and y1 define the upper left corner and x2 and y2 the lower right corner of the clipping area.

This is not real clipping, since the lines which are too long are only broken up by TCLIP (provided that WRAP ON is active) and the output below the bottom edge causes the area to scroll up (provided that SCROLL ON is active). TCLIP OFF removes these limitations.

**Example:**

```
SCREEN 3
PRINT STRING$("-",60)
TCLIP 0,0,50,30
PRINT STRING$("-",60)
TCLIP OFF
PRINT STRING$("-",60)
REPEAT
UNTIL LEN(INKEY$)
```

**Remarks:** -

**See**

**Also:** TCLIP, TGET, TPUT, TBOX, TPBOX, TCOLOR

## TCOLOR Command

**Action:** sets the character attribute for writing in text mode.

**Syntax:** TCOLOR *n*  
*n*: *iexp*

**Abbreviation:** tco *n*

**Explanation:** TCOLOR *n* sets the character attribute for writing in text mode (PRINT). *n* refers to the byte which, depending on the resolution, describes the character attributes.

**For MDA resolution this byte is as follows**

Bit	7	6	5	4	background
	1	x	x	x	blinking
	0	x	x	x	steady
	x	0	0	0	black background
	x	1	1	1	white background

Bit	3	2	1	0	foreground
	1	x	x	x	high intensity
	0	x	x	x	normal intensity
	x	0	0	0	black characters
	x	0	0	1	underlined
	x	1	1	1	white characters

**For standard CGA, EGA and VGA resolutions:**

Bit	7	6	5	4	background
	1	x	x	x	blinking
	0	x	x	x	steady
	x	0	0	0	black
	x	0	0	1	blue
	x	0	1	0	green
	x	0	1	1	cyan

## Commands and functions

---

	x	1	0	0	red
	x	1	0	1	magenta
	x	1	1	0	brown
	x	1	1	1	white
Bit	3	2	1	0	foreground
	1	x	x	x	high intensity
	0	x	x	x	normal intensity
	0	0	0	0	black
	0	0	0	1	blue
	0	0	1	0	green
	0	0	1	1	cyan
	0	1	0	0	red
	0	1	0	1	magenta
	0	1	1	0	brown
	0	1	1	1	white
	1	0	0	0	grey
	1	0	0	1	light blue
	1	0	1	0	light green
	1	0	1	1	light cyan
	1	1	0	0	pink
	1	1	0	1	purple pink
	1	1	1	0	yellow
	1	1	1	1	bright white

It's recommended that `n` be specified in binary or hexadecimal since this is much easier to see. `n=%10001011=$8B` will, therefore, set the blinking character to light cyan on black background.

### Example:

```
SCREEN 3
TCOLOR %00000100
TBOX 2,2,2,51,20
TCLIP 3,3 TO 50,19
a$="First, it is slightly cheaper; "
b$="and second it has the words "
c$="DON'T PANIC inscribed in large "
```

```
d$="friendly letters on its cover"  
PRINT  
PRINT  
PRINT a$;b$;c$;d$  
REPEAT  
UNTIL LEN(INKEY$)
```

```
// Writes a quote from D. ADAMS', The Hitchhiker's  
// Guide to the Galaxy, Pocket Books, N.Y, 1981 to  
// the screen.
```

**Remarks:**

Since the commands TBOX and TPBOX create frames out of graphic characters in the IBM character set, TCOLOR will influence the colour of these frames as well.

To differentiate the frames from the text, set TCOLOR before using the TBOX or TPBOX commands and then, before printing the text, set the colour again by using another TCOLOR.

**Example:**

```
SCREEN 3  
TCOLOR %00000100  
TBOX 2,2,2,51,20  
TCLIP 3,3 TO 50,19  
TCOLOR %10001011  
a$="First, it is slightly cheaper; "  
b$="and second it has the words "  
c$="DON'T PANIC inscribed in large      "  
d$="friendly letters on its cover"  
PRINT  
PRINT  
PRINT a$;b$;c$;d$  
REPEAT  
UNTIL LEN(INKEY$)
```

```
// Writes the above text within a black TBOX using  
// blinking cyan characters.
```

## Commands and functions

---

**See**

**Also:**

COLOR, SYSCOL



## TEXT Graphic command

**Action:** output of an expression as graphic text

**Syntax:** TEXT *x,y,exp*  
*x,y:* *iexp*  
*exp:* *aexp* or *sexp*

**Abbreviation:** te *x,y,exp*

**Explanation:** TEXT *x,y,exp* prints expression *exp* as graphic text at coordinates *x,y*. The point defined with *x,y* is aligned with the left corner of the base line of the first character in *exp*.

**Example:**

```
SCREEN 16                                EGA mode
s$="Test Test Test"
FOR i%=0 TO 10
    TEXT 50,ADD(SHL(i%,4),16),s$
NEXT i%

// Writes Test Test Test in different ways to the
// screen.
```

**Remarks:** -

**See**  
**Also:** -

### TGET Command

**Action:** cuts out a rectangular area in text mode.

**Syntax:** TGET x1,y,x2,y2,a\$  
x1,y1,x2,y2: *iexp*  
a\$: *svar*

**Abbreviation:** tge x1,y,x2,y2,a\$

**Explanation:** TGET x1,y,x2,y2,a\$ reads an area of text into a string variable. The area is defined with x1 and y1 for the upper left corner, and x2 and y2 for the lower right corner.

**Example:**

```
SCREEN 3
TBOX 2,1,1,40,10
a$="The Encyclopedia Galactica defines"
b$="a robot as a mechanical apparatus"
c$="to do the work of a man."
d$="The marketing division of the Sirius"
e$="Cybernetics Corporation defines a"
f$="robot as 'Your Plastic Pal Who's"
g$="Fun to Be With.'"
PRINT
PRINT a$
PRINT b$
PRINT c$
PRINT d$
PRINT e$
PRINT f$
PRINT g$
REPEAT
UNTIL LEN(INKEY$)
//
TGET 1,1,40,10 a$
//
```

```
TPUT 1,12,a$  
REPEAT  
UNTIL LEN(INKEY$)
```

**Remarks:** The area "cut out" with TGET x1,y1,x2,y2,a\$ can be re-inserted with TPUT x,y,a\$.

**See**

**Also:** TCLIP, TCLIP OFF, TBOX, TPBOX, TPUT, TCOLOR

## TIME\$ Function

**Action:** returns the system time.

**Syntax:** TIME\$

**Explanation:** TIME\$ returns the system time in the following format:

HH:MM:SS (Hours:Minutes:Seconds)

**Example:** PRINT TIME\$ // prints the system time

**Remarks:** -

**See**

**Also:** DATE\$

## TIME\$ = Command

**Action:** sets the system time.

**Syntax:** TIME\$ = time\$  
*time\$:* *sexp*

**Explanation:** TIME\$ = time\$ sets the system time in the following format:

HH:MM:SS (Hours:Minutes:Seconds)

**Example:** TIME\$ = "23:50:39"

**Remarks:** The setting of both the system date and the system time can be done with the SETTIME command.

**See**

**Also:** DATE\$ =, SETTIME

# TIMER Function

**Action:** returns the time in milliseconds.

**Syntax:** TIMER

**Abbreviation:** -

**Explanation:** Timer returns the time in milliseconds. The internal resolution amounts only to 1/18.2 seconds.

**Example:**

```
t%=TIMER
FOR i%=1 TO 10000
NEXT i%
PRINT (TIMER-t%)/1000

// Returns the time required by the empty loop in
// seconds.
```

**Remarks:** -

**See**

**Also:** -

## TITLEW# Graphic command

**Action:** writes a string on the title line of a window.

**Syntax:** TITLEW #n,a\$  
*n:* *iexp*  
*a\$:* *sexp*

**Abbreviation:** -

**Explanation:** TITLEW #n,a\$ writes the string a\$ on the title line of the window n, previously opened with OPENW #n. n can have the values of 1, 2, 3 or 4.

**Example:**

```
SCREEN 16                // EGA mode
TITELW #1," GFA-BASIC window "
OPENW #1,10,10,200,100,-1
REPEAT
    a$=INKEY$
UNTIL a$=CHR$(27)
CLOSEW #1
SCREEN 3
EDIT

// Draws a window on the screen and writes "GFA-BASIC
// window" on the title line of the window.
```

**Remarks:** -

**See**

**Also:** OPENW, CLOSEW, INFOW, SIZEW, TOPW,  
FULLW, CLEARW

### TOPW# Command

**Action:** activates a window.

**Syntax:** TOPW #n  
*n:* *iexp*

**Abbreviation:** -

**Explanation:** When several windows are opened, TOPW #n activates the window (1, 2, 3 or 4) specified in n. If this window is covered by another, it is brought up to the front.

**Example:**

```
SCREEN 16                // EGA mode
OPENW #1,10,10,200,100,-1
OPENW #2,15,15,200,100,-1
OPENW #3,20,20,200,100,-1
OPENW #4,25,25,200,100,-1
TOPW #2
KILLEVENT
CLS
REPEAT
    a$=INKEY$
UNTIL a$=CHR$(27)
```

```
// Draws four windows one after the other on the
// screen and then activates the window number 2
// without responding to redraw events which aren't
// necessary here.
```

**Remarks:** -

**See**

**Also:** OPENW, CLOSEW, TITLW, INFOW, SIZEW,  
FULLW, CLEARW



## TOUCH Command

**Action:** updates the time and date stamps of a file with current values.

**Syntax:** TOUCH[#]n  
*n: iexp*

**Abbreviation:** tou[#]n

**Explanation:** TOUCH[#]n works only on files already opened with OPEN by making their time and date stamps current. The time and date stamps of the open file are set to values obtained from the system clock.

**Example:**

```
OPEN "o",#1,"TEST.TXT
FOR i%=1 TO 20
  PRINT #1, STR$(i%)
NEXT i%
CLOSE #1
FILES "TEST.TXT"
//
DELAY 20           // a 20 second pause
//
OPEN "u",#1,"TEST.TXT"
TOUCH #1
CLOSE #1
FILES "TEST.TXT"

// Opens the TEST.TXT file and writes the numbers
// from 1 to 20 to it. The FILES is then used to
// print, among others, the time and date stamps.
// A 20 second pause follows next. The time and date
// stamp of the TEST.TXT file are then updated with
// TOUCH and printed again using FILES.
```

## Commands and functions

---

Remarks: -

See

Also:

## TPBOX Command

**Action:** draws a rectangle in text mode using the graphic characters from the IBM character set.

**Syntax:** TPBOX *n,x,y,w,h*  
*n,x,y,w,h:* *iexp*

**Abbreviation:** tpb *n,x,y,w,h*

**Explanation:** TPBOX *n,x,y,w,h* draws a frame, using the relevant characters from the IBM characters set, whose upper left corner is defined in *x* and *y*, and whose width and height are defined in *w* and *h*.

In contrast to TBOX, TPBOX fills the inside area with, optionally coloured, space characters.

The style of the frame is defined with *n* as follows:

*n* = 0 build a frame out of space characters  
*n* = 1 build a frame out of single lines  
*n* = 2 build a frame out of double lines

**Example:**

```
SCREEN 3
TCOLOR $1E // yellow on blue
TCLIP 3,3 TO 50,19
TPBOX 2,2,2,51,20
a$="First, it is slightly cheaper; "
b$="and second it has the words "
c$="DON'T PANIC inscribed in large "
d$="friendly letters on its cover"
PRINT
PRINT
PRINT a$;b$;c$;d$
REPEAT
UNTIL LEN(INKEY$)
```

## Commands and functions

---

```
// Writes a quote from D. ADAMS', The Hitchhiker's  
// Guide to the Galaxy, Pocket Books, N.Y, 1981 to  
// the screen.
```

**Remarks:** -

**See**

**Also:** TCLIP, TCLIP OFF, TBOX, TPUT, TGET,  
TCOLOR

## TPUT Command

**Action:** restores a rectangular area in text mode.

**Syntax:** TPUT *x,y,a\$*  
*x,y:* *iexp*  
*a\$:* *svar*

**Abbreviation:** tpu *x,y*

**Explanation:** TGET *x1,y,x2,y2,a\$* reads an area of text into a string variable. The area is defined with *x1* and *y1* for the upper left corner, and *x2* and *y2* for the lower right corner.

The area which was "cut out" can then be re-inserted with TPUT *x,y,a\$*.

**Example:**

```
SCREEN 3
TBOX 2,1,1,40,10
a$="The Encyclopedia Galactica defines"
b$="a robot as a mechanical apparatus"
c$="to do the work of a man."
d$="The marketing division of the Sirius"
e$="Cybernetics Corporation defines a"
f$="robot as 'Your Plastic Pal Who's"
g$="Fun to Be With.'"
PRINT
PRINT a$
PRINT b$
PRINT c$
PRINT d$
PRINT e$
PRINT f$
PRINT g$
REPEAT
UNTIL LEN(INKEY$)
```

## Commands and functions

---

```
//  
TGET 1,1,40,10 a$  
//  
TPUT 1,12,a$  
//  
REPEAT  
UNTIL LEN(INKEY$)
```

**Remarks:**

-

**See**

**Also:**

TCLIP, TCLIP OFF, TBOX, TPBOX, TGET,  
TCOLOR

## TRACE\$ Variable

**Action:** locates errors in a program.

**Syntax:** -

**Explanation:** TRACE\$ is a string variable which, following the TRON procedurename, contains the command which will be executed next. TRON procedurename, specifies a PROCEDURE which will be invoked before execution of every command which follows the TRACE\$. The combination of TRON procedurename and TRACE\$ is a very efficient way of looking for errors.

**Example:**

```
SCREEN 16                // EGA mode
GRAPHMODE 3              // XOR
LINE 0,360,639,360
TRON debug
DO UNTIL MOUSEK AND 2
  a$=INKEY$
  IF MOUSEK AND 1
    IF MOUSEY<330
      BOX MOUSEX,MOUSEY,ADD(MOUSEX,60),ADD(MOUSEY,30)
      WHILE MOUSEK
        WEND
      ENDIF
    ENDIF
  LOOP
SCREEN 3                  // Text mode
EDIT
//
PROCEDURE debug
  IF a$=CHR$(27)
    PRINT AT(10,24);SPACE$(60)
    PRINT AT(10,24);LEFT$(TRACE$,60)
    PAUSE 20
```

## Commands and functions

---

```
ENDIF  
RETURN
```

```
// Draws on the screen when the left mouse button is  
// held down. When the Esc key is pressed TRACE$  
// shows the commands on the bottom of the screen.  
// The program is terminated by pressing the right  
// mouse button.
```

**Remarks:** -

**See**

**Also:** DUMP, TRON, TROFF



## TRIM\$ String command

**Action:** removes spaces at the beginning or end of a string expression.

**Syntax:** TRIM\$(a\$)  
*a\$: sexp*

**Abbreviation:** -

**Explanation:** -

**Example:**

```
a$=" Hello GFA "
```

```
PRINT a$           // prints ' Hello GFA '
```

```
PRINT TRIM$(a$)    // prints 'Hello GFA'
```

**Remarks:** -

**See**

**Also:** -

## TROFF Command

**Action:** turns off TRON.

**Syntax:** TROFF

**Abbreviation:** trof

**Explanation:** see TRON

**Example:** see TRON

**Remarks:** -

**See**  
**Also:** DUMP, TRON, TRACES

## TRON Command

**Action:** lists commands during program execution.

**Syntax:** TRON[#n]  
*n:* *iexp*

**Abbreviation:** -

**Explanation:** The TRON command (TRACE ON) causes each command that follows it to be shown on the screen. By specifying a channel number this list can be redirected to a file previously opened with OPEN "o", #n,"filename".

TRON can be followed by the name of a PROCEDURE which is executed before each command. This special case is described in "TRACE\$ Variable".

**Example:**

```
PRINT "Test program"
TRON
FOR i%=1 TO 5
    PRINT SIN(i%)
NEXT i%
TROFF
PRINT "Program end"
//
// or
//
PRINT "Test program"
OPEN "o",#1,"C:\TEST.DAT"
TRON #1
FOR i%=1 TO 5
    PRINT #1, SIN(i%)
NEXT i%
TROFF
CLOSE #1
```

## Commands and functions 8.2.10

---

```
PRINT "Program end"

// prints
//
// Test program
// FOR i%=1 TO 5
// PRINT #1,SIN(i%)
// 0.8414709848079
// NEXT i%
// PRINT #1,SIN(i%)
// 0.9092974268257
// NEXT i%
// PRINT #1,SIN(i%)
// 0.1411200080599
// NEXT i%
// PRINT #1,SIN(i%)
// -0.7568024953079
// NEXT i%
// PRINT #1,SIN(i%)
// -0.9589242746631
// NEXT i%
// Program end
//
// on the screen or saves it in the file 'TEST.DAT'
// in partition C.
```

**Remarks:** The TROFF turns the TRON off.

**See**

**Also:** DUMP, TROFF, TRACE\$

## TRUE Variable

**Action:** -1 constant

**Syntax:** TRUE

**Explanation:** Contains the value for logical true.

**Example:**

```
i%=20
IF i%
    a!=TRUE
    PRINT "i% is not equal to 0; a!=";a!
ENDIF
i%=0
IF !i%
    a!=FALSE
    PRINT "i% is equal to 0; a!=";a!
ENDIF

// prints
// i% is not equal to 0; a!=-1
// i% is equal to 0; a!=0
```

**Remarks:** -

**See**

**Also:** FALSE

## TRUNC() Numeric function

**Action:** returns the integer part of a numeric expression.

**Syntax:** TRUNC(*x*)  
*x*: *aexp*

**Explanation:** see FIX

**Example:**

```
PRINT TRUNC(3*1.2) // prints 3
PRINT TRUNC(3*-1.2) // prints -3
PRINT FIX(3*1.2) // prints 3
PRINT FIX(3*-1.2) // prints -3
```

**Remarks:** FIX() or INT() are synonymous with TRUNC() and can be used instead.

**See**

**Also:** FLOOR(), INT(), CEIL(), FIX(), FRAC()

## TYPE Command

**Action:** DEFINITION of a file structure

**Syntax:** TYPE name:  
.  
.  
.  
ENDTYPE  
*name: an ASCII character string*

**Abbreviation:** -

**Explanation:** TYPEs, called 'Records' in Pascal or 'Structures' in C, are used to group data which belongs together. A Type comprises the data structure in which all elements of this type are defined. By defining the elements the types are also defined. A TYPE can then be assigned a TYPE variable.

A TYPE always ends with a colon ":", while a TYPE variable always ends with a point ".". The TYPE elements are defined with a variable category and a name. For example

TYPE window:

```
- CHAR*20    title$  
- CHAR*20    info$  
- UBYTE      nr  
- CARD       xp  
- CARD       yp  
- CARD       w  
- CARD       h  
- BYTE       s  
ENDTYPE
```

Do note that the name of the TYPE is not in quotes, that it ends with a colon ":" and that the variable types are prefixed with a dash "-".

The above example defines a TYPE called window. Certain elements which can be used with the GFA-BASIC window are then defined.

A 20 character string called title\$ is defined first (window title, for example). Another 20 character string called info\$ is defined next (info line, for example). The UBYTE variable takes the number of the window. The following four CARD variables n take the window coordinates (x and y position, width and height). The BYTE variable stores the window type (3D-effect, for example).

The following statement

```
window: f1. or DIM window: f1.
```

creates the TYPE variable f1 with TYPE window: while ERASE f1. deletes it.

The following variable types are allowed within a GFA-BASIC TYPE:

- INT            32 bit integer (postfix %)
- LONG          32 bit integer (postfix %)
- WORD          16 bit integer (postfix &)
- SHORT        16 bit integer (postfix &)
- CARD          16 bit unsigned integer
- UWORD        16 bit unsigned integer
- USHORT       16 bit unsigned integer
- BYTE          8 bit integer (postfix |)
- UBYTE        8 bit unsigned integer
  
- CHAR(n)      a string with n characters
- CHAR\*n      a string with n characters
- STRING(n)    a string with n characters



- **STRING\*n**     a string with n characters
- **DOUBLE**     floating point variable in IEEE double format
- **SINGLE**     floating point variable in IEEE single format

The variable names of **CHAR** and **STRING** elements must end with a "\$", while the **DOUBLE** and **SINGLE** element names must end with a "#".

The elements in a **TYPE** variable can be assigned a value or an expression. In this way, the assignments

```
f1.title$="Figure 1"  
f1.info$="Lissajous"  
f1.nr=1  
f1.xp=0  
f1.yp=0  
f1.w=200  
f1.h=100  
f1.s=-1
```

assign certain values and expressions to the elements in the **TYPE** variable f1.

Following that a

```
OPENW #f1.nr,f1.xp,f1.yp,f1.w,f1.h,f1.s  
TITLEW#f1.nr,f1.title$  
INFOW #f1.nr,f1.info$
```

can then open a GFA-BASIC window in graphic mode.

The **LEN** function can also be used on a **TYPE**. For example, **LEN(window:)** returns the value 50. In this way you can determine how much memory a **TYPE** requires.

The LEN function can be used on a TYPE variable as well. In the above example LEN(f1.) returns the value 50. Furthermore, the functions VARPTR(), ARRPRTR(), V: and \* can also be used on a TYPE variable. They all return the addresses of TYPE variables:

```
PRINT VARPTR(f1.)
PRINT ARRPRTR(f1.)
PRINT V:f1.
PRINT *f1.
```

In this way it is possible to move the contents of one TYPE variable to another of the same TYPE.

For example, the TYPE variable f is created with

```
window:f.
```

The contents of f1 can then be assigned with the TYPE variable f. as follows

```
f.=f1.
```

Following that a

```
OPENW #f.nr,f.xp,f.yp,f.w,f.h,f.s
TITLEW #f.nr,f.titel$
INFOF #f.nr,f.info$
```

will then, just like in the original example, open a window using f.

Finally, the pointer operations can also be used on TYPEs and TYPE variables, for example

```
t.=type:{addr}          TYPE-PEEK
type:{addr}=t.          TYPE-POKE
type:{addr1}=type:{addr2} TYPE-MOVE
```

The following operations can be used on TYPE variable elements:

t.var=27	direct assignment
t.var=a%	indirect assignment
t.var\$="Hello"	direct assignment
t.var\$=a\$	indirect assignment
t.var ++	increment
t.var=h.var+k.var	indirect assignment
t.var#=7	multiplication
z=t.var#	reverse assignment
PRINT t.var\$	output

**Example:**

```
TYPE dta:
- CHAR*21  fs_donttouch$
- BYTE     fs_attr
- CARD     fs_time
- CARD     fs_date
- LONG     fs_size
- CHAR*14  fs_name$
ENDTYPE
dta%=FGETDTA()
e%=FSFIRST("*.\"",17)
WHILE e% >= 0
  a$=SPACE$(60)
  MID$(a$,2)={dta%}.fs_name$
  q$="      "
  IF BTST({dta%}.fs_attr,4)
    RSET q$="<DIR>"
  ELSE
    RSET q$=str$({dta%}.fs_size)
  ENDIF
  MID$(a$,16)=q$
  a%={dta%}.fs_date
  q$=DEC$(a% and 31,2)+"."+DEC$((a% >> 5) & 15,2)
  q$=q$+"."+DEC$((a% >> 9) + 1980,4)
  MID$(a$,30)=q$
```

## Commands and functions

---

```
a%={dta%}.fs_time
q$=DEC$(a% >> 11,2)+". "+DEC$((a% >> 5) & 63,2)
q$=q$+" "+DEC$((a% & 31) * 2,2)
MID$(a$,42)=q$
PRINT a$
REPEAT
UNTIL LEN(INKEY$)
e%=FSNEXT()
WEND
PRINT e%
```

```
// This example defines a TYPE, which corresponds to
// the structure of a DTA (see FGETDTA()).
// The address of the DTA is then obtained with
// dta%=FGETDTA() and its contents are read using the
// relevant TYPE PEEKs and displayed on the screen.
// The string operations q$=... are only necessary,
// to format the relevant information.
// When the program starts the current directory in
// displayed on the screen.
```

**Remarks:**

-

**See**

**Also:**

-



## UCASE\$() String function

**Action:** converts all lower case letters of a string expression to capital letters, with the exception of PC umlauts (US character table).

**Syntax:** UCASE\$(a\$)  
*a\$: sexp*

**Abbreviation:** -

**Explanation:** -

**Example:** PRINT UCASE\$("Hello GFA, öööha")

```
// prints  
// HELLO GFA öööHA
```

**Remarks:** -

**See**

**Also:** UPPER\$(), LOWER\$(), LCASE\$(), XLATE\$()

## UPPER\$() String function

**Action:** converts all lower case letters of a string expression including all PC umlauts (IBM character set) to capital letters.

**Syntax:** UPPER\$(a\$)  
*a\$:* *sexp*

**Abbreviation:** -

**Explanation:** -

**Example:** PRINT UPPER\$("Hello GFA, ööööha")

```
// prints  
// HELLO GFA ÖÖÖÖHA
```

**Remarks:** -

**See**

**Also:** LOWER\$(), UCASE\$(), LCASE\$(), XLATE\$()

# USHORT() Function

**Action:** performs AND 65535.

**Syntax:** USHORT(*m*)  
*m*: *exp*

**Explanation:** Limits *m* to 16 bits by clearing the bits 16 to 31. Performs AND 65536.

**Example:**

```
s%=32
m%=2096
a=100000
PRINT BIN$(m%,s%)
PRINT BIN$(USHORT(m%),s%)
PRINT STRING$(s%,"-")
PRINT BIN$(a,s%)
PRINT BIN$(USHORT(a),s%)
PRINT BIN$(a AND 65535,s%)
PRINT BIN$(a MOD 65536,s%)
PRINT USHORT(a)
```

```
// prints
//
// 00000000000000000000100000110000
// 00000000000000000000100000110000
// -----
// 00000000000000011000011010100000
// 00000000000000001000011010100000
// 00000000000000001000011010100000
// 0000000000000001000011010100000
// 34464
//
```



**Remarks:** CARD() is synonymous with USHORT() and can be used instead.

**See**

**Also:** BYTE(), WORD(), CARD(), SHORT()

### USHORT{} Function

**Action:** reads a word (16 bits) from an address.

**Syntax:** USHORT{addr}  
*addr: address*

**Abbreviation:** -

**Explanation:** Reads a word (16 bits) from an address.

**Example:** PRINT USHORT{{\*a\$}+4} // prints LEN(A\$)  
  
PRINT USHORT{\$40:\$1E} // prints the first word at  
// offset \$1E in segment \$40.

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

CARD{} and UWORD{} are synonymous with USHORT{} and can be used instead.

addr: see the {} function.

**See**

**Also:** BYTE{}, WORD{}, CARD{}, INT{}, LONG{}, {}, SINGLE{}, DOUBLE{}, SHORT{}, UWORD{}

## USHORT{} = Command

**Action:** writes a word (16 bits) to an address.

**Syntax:** USHORT{addr} = m  
*addr: address*  
*m: iexp*

**Abbreviation:** -

**Explanation:** Writes a word (16 bits) to an address.

**Example:** -

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

CARD{} = and UWORD{} = are synonymous with USHORT{} = and can be used instead.

addr: see the {} function.

**See**

**Also:** BYTE{} =, WORD{} =, INT{} =, CARD{} =, LONG{} =, {} =, SINGLE{} =, DOUBLE{} =, SHORT{} =, UWORD{} =

## UWORD() Function

**Action:** performs AND 65535

**Syntax:** UWORD(*m*)  
*m*: *iexp*

**Explanation:** Limits m to 16 bits by clearing the bits 16 to 31.

[illegible]

**Remarks:** CARD() and USHORT() are synonymous with UWORD() and can be used instead.

**See**  
**Also:** `BYTE()`, `WORD()`, `CARD()`, `SHORT()`, `UWORD()`

## UWORD{} Function

**Action:** reads a word (16 bits) from an address.

**Syntax:** UWORD{addr}  
*addr: address*

**Abbreviation:** -

**Explanation:** Reads a word (16 bits) from an address.

**Example:** -

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

CARD{} and USHORT{} are synonymous with UWORD{} and can be used instead.

addr: see the {} function.

**See**

**Also:** BYTE{}, WORD{}, CARD{}, INT{}, LONG{}, {}, SINGLE{}, DOUBLE{}, SHORT{}, USHORT{}

### UWORD{} = Command

**Action:** writes a word (16 bits) to an address.

**Syntax:** UWORD{addr} = m  
*addr: address*  
*m: iexp*

**Abbreviation:** -

**Explanation:** Writes a word (16 bits) to an address.

**Example:** -

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

CARD{} = and USHORT{} = are synonymous with UWORD{} = and can be used instead.

addr: see the {} function.

**See**

**Also:** BYTE{} =, WORD{} =, INT{} =, CARD{} =, LONG{} =, {} =, SINGLE{}, DOUBLE{} =, SHORT{} =, USHORT{} =



## V: Function

**Action:** returns, in case of strings, the address of the string itself (not the descriptor address) or, in case of arrays, the address of an array element or, in case of simple variables, the address of the variable.

**Syntax:** V:x  
x: *name of a variable of any type*

**Explanation:** Returns = > addr

**Example:**

```
PRINT V:a$           // prints the address of a$.
PRINT V:a(3)         // prints the address of
//                  a(3).
PRINT V:a%           // prints the address of a%.
```

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

VARPTR() is synonymous with V: and can be used instead.

addr: see the {} function.

**See**  
**Also:** ARRPTR(x), PEEK(), DPEEK(), LPEEK(),  
VARPTR()



## VAL() Function

**Action:** converts a string expression into a number.

**Syntax:** VAL(*a\$*)  
*a\$*: *sexp*

**Explanation:** If during conversion VAL() encounters a character which cannot be interpreted as a part of a number ("1234a" for example), the evaluation of the string expression is terminated. The number obtained up until this point (1234 in the above example) is then returned by VAL().

If the string expression begins with a character which cannot be interpreted as a part of a number, VAL() returns 0.

If the string expression begins with &X or %, the binary conversion takes place. &O or &Q converts to octal, while &H, & or \$ converts to hexadecimal.

**Example:**

```
PRINT VAL("-.123") // prints -0.123
a$=STR$(12345)
PRINT VAL(a$) // prints 12345
PRINT VAL("&H"+"AF") // prints 175
PRINT VAL("$AA") // prints 170
PRINT VAL("%10101011) // prints 171
```

**Remarks:** -

**See**

**Also:** STR\$(), VAL?()

### VAL?() Function

**Action:** determines how many places of a string expression will be converted when using VAL().

**Syntax:** VAL?(a\$)

*a\$: sexp*

**Explanation:** VAL?(a\$) returns 0, if a\$ contains no characters which can be interpreted as numbers.

**Example:**

```
PRINT VAL?("12345")      // prints 5
x=VAL?("3.00 DM")
PRINT x                  // prints 4
PRINT VAL?("Hello GFA") // prints 0
```

**Remarks:** -

**See**

**Also:** VAL()

## VAR Parameter

**Action:** declaration part of the parameter list for a PROCEDURE or FUNCTION, which is followed Call by Reference Variables.

**Syntax:** PROCEDURE name([a,b,...] VAR x,y,..a(),b(),...)

<i>name:</i>	<i>procedure or function name</i>
<i>a,b,...:</i>	<i>aexp, sexp; Call by Value variables</i>
<i>x,y,...:</i>	<i>Call by Reference variables</i>
<i>a(),b(),...:</i>	<i>pointer to arrays</i>

**Explanation:** When the main program calls a subroutine, it is often necessary to pass variables from the main routine to this subroutine. When the subroutine is invoked, these variables are contained in a list and the called subroutine must declare them in its parameter list.

Two categories of these (call-) variables are available:

1. Variables, passed to the subroutine, which are changed in the subroutine but this change need not be preserved for the rest of the program. In this case the subroutine only needs the value of the relevant variable. The subroutine uses these variables without changing the actual variable. Such variables are described as Call by Value variables. They are used as 'local' variables in the subroutine.

Since instead of the actual variables only their values are used in the subroutine, their names must not be listed in the parameter list of the subroutine. When the variable type is the same it is sufficient to supply only the place holder.

**Example:**

```
a%=15
@test(a%)
PRINT a%''c%
```

```
REPEAT
UNTIL LEN(INKEY$)
//
PROCEDURE test(c%)
  c%*=3
  PRINT "3*15 = ";c%
RETURN

// prints
// 3*15 = 45
// 3 0
```

2. Variables, passed to the subroutine, which are changed in the subroutine and this change is preserved for the rest of the program. In this case the subroutine gets the pointers to the relevant variables. The subroutine then works with the values in the actual variables themselves. Such variables are described as Call by Reference variables. They are used as 'global' variables in the subroutine.

Since instead of the actual variables only their pointers are used in the subroutine, their names must not be listed in the parameter list of the subroutine. When the variable type is the same it is sufficient to supply only the place holder. In order to indicate to the subroutine where in the parameter list the Call by Reference variables are, they are prefixed with a VAR!

### Example:

```
a%=15
@test(a%)
PRINT a%'c%
REPEAT
UNTIL LEN(INKEY$)
//
```

```
PROCEDURE test(VAR c%)
  c%*=3
  PRINT "3*15 = ";c%
RETURN

// prints
// 3*15 = 45
// 45  0
```

**Remarks:**

Call by Value and Call by Reference variables can be used together in the same subroutine. The variable (or parameter) list must start with Call by Value variables and be followed by Call by Reference variables.

Since Call by Reference variables pass only the pointer to the corresponding variables they can also be used to pass the whole arrays to the subroutine.

**Example:**

```
n%=10
DIM a%(n%,n%)
x%=0
FOR i%=1 TO n%
  FOR j%=1 TO n%
    a%(i%,j%)=j%
    PRINT a%(i%,j%)
  NEXT j%
  PRINT
NEXT i%

PROCEDURE test(f% VAR b%(),x%)
  LOCAL i%,j%
  FOR i%=1 TO f%
    FOR j%=1 TO f%
      b%(i%,j%)*=10
      x%+=b%(i%,j%)
    NEXT j%
  NEXT i%
RETURN
```

## Commands and functions

---

See

Also:

-

## Variable types

Name	Postfix	Memory requirements	Variable type
Double	#	8 bytes	floating point
Long	%	4 bytes	integer (signed)
Word	&	2 bytes	integer (signed)
Byte		1 byte	integer (unsigned)
Boolean	!	1 bit	integer (signed)
String	\$	depends on string length	(32767 characters maximum)

## Double #

The floating point variable `Double` occupies 8 bytes (64 bits) of memory. The postfix `"#"` is assumed as default for all variables. This applies only when the variable type is not defined explicitly using `DEFBIT`, `DEFBYT`, `DEFDBL`, `DEFINT`, `DEFLNG`, `DEFWRD`, `DEFSTR` or by attaching a postfix of a specific variable type. The range of available numbers conforms to the IEEE double format and is defined as real numbers between  $2.2\text{E}-308$  and  $1.67\text{E}+308$ .

**Example:** `x=123.337` or `x#=123.337`

### Long %

The signed integer Long occupies 4 bytes (32 bits) of memory. The range of valid numbers for variables with postfix "%" is whole numbers between -2147483648 and +2147483647.

**Example:** `x%=103124`

### Word &

The signed integer Word occupies 2 bytes (16 bits) of memory. The range of valid numbers for variables with postfix "&" is whole numbers between -32768 and +32767.

**Example:** `x&=12345`

### Byte |

The unsigned variable type Byte occupies 1 byte (8 bits) of memory. The range of valid numbers for variables with postfix "|" is natural numbers between 0 and 255.

**Example:** `x|=209`



## Boolean !

The signed variable type Boolean occupies 1 bit of memory. The variables with postfix "!" can only take the values -1 (TRUE) and 0 (FALSE). If a value other than zero is assigned to this variable, it always takes the value of -1, otherwise it defaults to 0. If a logical operation is false this variable takes the value of 0, and if the result is true it takes -1.

**Example:**

```
x!=FALSE // returns 0
x!=10>20 // returns 0
x!=-100 // returns -1
```

## String \$

The memory occupied by strings depends on the number of characters in each string. The spaces are counted as well! The maximum length of variables with postfix "\$" is 32767 characters.

Since the characters are encoded in ASCII with values between 0 and 255, each character in the string occupies 1 byte of memory.

Every string is defined with the help of a so-called descriptor. A string requires, first of all, 4 bytes of memory for a pointer to a specific memory address. This address contains the address of the descriptor (backtrailer) which occupies 4 bytes of memory. These are followed by 2 bytes (for GFA-BASIC MS-DOS and GFA-BASIC OS/2) or 4 bytes (for GFA-BASIC), which contain the string length. After that follows the string itself. If the number of characters in the string is odd, the string is padded with a filler byte (for GFA-

## Commands and functions

---

BASIC MS-DOS and GFA-BASIC OS/2) or with as many bytes as needed (up to 3), so that the total number of characters in the string is divisible by four (for GFA-BASIC).

**Example:**

x\$="Hello GFA"

needs

20 bytes:

4 bytes for the pointer,

4 bytes for the address of  
the descriptor,

2 bytes for the string  
length,

9 bytes for 9 characters,  
plus

1 byte filler.

The address of any variable can be obtained with functions VARPTR or V:(= variable pointer) and ARRPTR or \* (= array-pointer).

## VARIAT() Numeric function

**Action:** returns the number of permutations of  $n$  elements to  $k$ -th order without repetition.

**Syntax:** VARIAT( $n,k$ )  
 $n,k$ : *iexp*

**Explanation:** VARIAT( $n,k$ ) is defined as:  
$$\text{VARIAT}(n,k) = n! / (n-k)!$$

**Example:** PRINT VARIAT(6,2)      // prints      30

**Remarks:** If  $k > n$  an error is reported.

**See**

**Also:** FACT(), COMBIN()

### VARPTR() Function

**Action:** When used with a string it returns the address of the string itself (not the address of its descriptor), when used with arrays it returns the address of an array element and when used with simple variables it returns the address of the variable.

**Syntax:** `VARPTR(x)`  
*x:* *name of any variable type*

**Abbreviation:** v:

**Explanation:** Returns = > addr

**Example:**

```
PRINT VARPTR(a$)      // prints the address of a$
PRINT VARPTR(a(3))    // prints the address of a(3)
PRINT V:a%             // prints the address of a%
```

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

addr: see the {} function.

**See**

**Also:** `ARRPTR()`, `PEEK()`, `DPEEK()`, `LPEEK()`

## VOID Command

**Action:** voids a numeric expression.

**Syntax:** VOID a  
*a:* *aexp*

**Abbreviation:** VO a

**Explanation:** VOID causes a calculated value or a value returned from a function, to be indeed calculated, but because of VOID it's immediately "forgotten".

**Example:**

```
FOR i%=1 TO 30
  VOID SIN(i%)
  PRINT i%
NEXT i%           // prints 1...30 very slowly
```

**Remarks:** -

**See**  
**Also:** ~

### VTAB Command

**Action:** cursor positioning

**Syntax:** VTAB row  
*row: iexp*

**Abbreviation:** -

**Explanation:** places the cursor on the line specified in row.

**Example:** PRINT AT(1,1);"Hello GFA"  
VTAB 4  
PRINT "Hello GFA"

```
// Prints Hello GFA from the first column of the  
// first line and then again on the fourth line.
```

**Remarks:** -

**See**

**Also:** LOCATE, PRINT AT, HTAB, TAB



### WHILE...WEND Structure

**Action:** A terminal program loop which runs until the condition at the beginning of the loop is logically "true".

**Syntax:**

```
WHILE condition
    // programsegment
WEND
```

*condition: any numeric, logical or string condition*

**Abbreviation:**

```
whi condition
    // programsegment
we
```

**Explanation:** The start of a WHILE...WEND loop must contain a numeric, logical or string condition, which is evaluated before each execution of the body of the loop. If the condition is logically "true", the body of the loop is executed. Otherwise, a branch is taken to the program statement immediately after WEND.

The WHILE...WEND loop is an entry tested loop. This means that the loop executes only when the condition at the beginning of the loop is logically "true".

By using an "EXIT...IF" command, the WHILE...WEND loop can be terminated regardless of whether the loop condition is fulfilled.

**Example:**

```
WHILE NOT UPPER$(INKEY$)="A"
    // programsegment
WEND
```

```
// A loop which runs as long as no lowercase or
// uppercase "a" is entered from the keyboard.
```



**Remarks:** The WHILE...WEND loop can be seen as a logical negation of the REPEAT...UNTIL loop, whereby a WHILE NOT corresponds to an UNTIL.

**See**

**Also:** FOR...NEXT, REPEAT...UNTIL, DO...LOOP

### WIN Command

**Action:** selects a window.

**Syntax:** WIN #*n*  
*n*: *iexp* (0,...4)

**Abbreviation:** -

**Explanation:** WIN #*n*, sets the internal variables for window with number *n*, for example clipping, \_X ,\_Y, PRINT output or WINDGET.

**Example:**

```
SCREEN 16
OPENW #1,100,100,200,200,-1
PRINT "Hello Window 1"
WIN #0
TEXT 50,50,STR$(MOUSEX,4)+STR$(MOUSEY,4)
WIN #1
PRINT "Window 1 again"
```

**Remarks:** -

**See**

**Also:** OPENW *n*

## WINDFIND Function

**Action:** determines the number of the window at a given coordinate point.

**Syntax:** WINDFIND *x,y,nr*  
*x,y:* *iexp*  
*nr:* *ivar*

**Explanation:** WINDFIND *x,y,nr* tests if there is a window at the position specified with the *x* and *y* coordinates. If there is, the number of the window at this position is returned in the variable *nr*. *nr* can take the values from 0 to 4, where 0 is the Desktop (window 0) and 1 to 4 correspond to "real" windows. If there are several windows on top of each other, *nr* returns the number of the window on the very top.

**Example:**

```
SCREEN 18
OPENW 1,10,10,300,200,-1
OPENW 2,50,50,300,200,-1
WINDFIND 15,15,a%
WINDFIND 60,60,b%
WINDFIND 340,240,c%
SCREEN 3
COLOR 1
PRINT a%'b%'c%
REPEAT
UNTIL LEN(INKEY$)
```

```
// Prints 1 2 2. The position 15,15 is on window #1
// only, while the position 340,240 on the window #2
// only. The position 60,60 is on both window #1 and
// window #2. Since OPENW #2 was after OPENW #1 the
// second window is the top window and WINDFIND
// returns in this case the value nr=2.
```

## Commands and functions

---

**Remarks:** -

**See**

**Also:** WINDGET, WINDSET

## WINDGET Command

**Action:** reads window parameters.

**Syntax:** WINDGET i,a[,b[,c...]]  
*i: iexp;*  
*a,b,c:var;*

**Explanation:** WINDGET reads various window related parameters.

i	Parameter
0	outer X-coordinate
1	outer Y-coordinate
2	outer width
3	outer height
4	inner X-coordinate
5	inner Y-coordinate
6	inner width
7	inner height
8 *	position of vertical slider (0..1000)
9 *	size of vertical slider (0..1000)
10 *	position of horizontal slider (0..1000)
11 *	size of horizontal slider (0..1000)
12	reads the window attributes (as set with OPENW)
13 *	reads the attributes of the pressed window button (from WINDSET)
14 *	character height (for example 8, 14, 16)
15 *	character set address
16	number of top window
17	number of second to top window
18	number of second to bottom window
19	number of bottom window

The asterisk indicates which parameters can be changed with WINDSET.

## Commands and functions

---

### Example:

```
SCREEN 16
OPENW #1,100,100,200,200,-1
vpos&=500,hpos&=500
WINDSET 8,vpos&,100,hpos&,100 // set slider position
//                               and size
DO
  GETEVENT
  SELECT MENU(1)
  CASE 7                          // Arrow Up
    vpos& -= 10
  CASE 8                          // Arrow Down
    vpos& += 10
  CASE 9                          // Arrow Left
    hpos& -= 10
  CASE 10                        // Arrow Right
    hpos& += 10
  CASE 11                        // Page Up
    vpos& -= 100
  CASE 12                        // Page Down
    vpos& += 100
  CASE 13                        // Page Left
    hpos& -= 100
  CASE 14                        // Page Right
    hpos& += 100
  CASE 15                        // vertical slider moved
    vpos& = MENU(7)
  CASE 16                        // horizontal slider moved
    hpos& = MENU(7)
ENDSELECT
IF MENU(1) >= 7 AND MENU(1) <= 16
  IF vpos& < 0
    vpos& = 0
  ELSE IF vpos& > 1000
    vpos& = 1000
  ENDIF
  IF hpos& < 0
    hpos& = 0
```

```
ELSE IF hpos& > 1000
    hpos& = 1000
ENDIF
WINDSET 8,vpos&
WINDSET 10,hpos&
ENDIF
LOOP
```

**Remarks:** -

**See**

**Also:** WINDSET

### WINDSET Command

**Action:** writes window parameters.

**Syntax:** WINDSET *i*,*a*[],*b*[],*c*...[]  
*i*,*a*,*b*,*c*: *iexp*;

**Explanation:** WINDSET writes various window related parameters.

<i>i</i>	Parameter
8	position of vertical slider (0..1000)
9	size of vertical slider (0..1000)
10	position of horizontal slider (0..1000)
11	size of horizontal slider (0..1000)
13	reads the window attributes of pressed window button (from WINDSET)
14	character height (for example 8, 14, 16)
15	character set address

**Example:** see WINDGET

**Remarks:** The position and size of a window can be modified with MOVEW and SIZEW, but the attributes can only be changed with CLOSEW/OPENW.

**See**

**Also:** WINDGET, MOVEW, SIZEW





## Commands and functions

---

**Remarks:**        `SHORT()` is synonymous with `WORD()` and can be used instead.

**See**  
**Also:**            `BYTE()`,   `CARD()`,   `SHORT()`,   `USHORT()`,  
                    `UWORD()`

## WORD{} Function

**Action:** reads a word (16 bits) from an address.

**Syntax:** WORD{addr}  
*addr: address*

**Abbreviation:** -

**Explanation:** Reads a word (16 bits) from an address.

**Example:** -

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

INT{} and SHORT{} are synonymous with WORD{} and can be used instead.

addr: see the {} function.

**See**

**Also:** DPEEK(), BYTE{}, CARD{}, INT{}, LONG{}, {}, SINGLE{}, DOUBLE{}, SHORT{}, USHORT{}, UWORD{}

### WORD{} = Command

**Action:** writes a word (16 bits) to an address.

**Syntax:** WORD{addr} = m  
*addr: address*  
*m: iexp*

**Abbreviation:** -

**Explanation:** Writes a word (16 bits) to an address.

**Example:** -

**Remarks:** An address is composed of a segment and an offset. To specify an address a 16-bit integer expression is given to represent the segment, followed by a colon and a 16-bit integer expression for the offset.

INT{} = and SHORT{} = are synonymous with WORD{} = and can be used instead.

addr: see the {} function.

**See**

**Also:** DPOKE(), BYTE{} =, CARD{} =, INT{} =, LONG{} =, {} =, SINGLE{} =, DOUBLE{} =, SHORT{} =, USHORT{} =, UWORD{} =

## WRAP ON and WRAP OFF Commands

**Action:** turns line wrap-around on and off.

**Syntax:** WRAP ON  
WRAP OFF

**Abbreviation:** -

**Explanation:** WRAP OFF turns off line wrap-around in text mode and for text output inside windows. The long lines are thereby broken up.

WRAP ON turns the line wrap-around back on.

**Example:**

```
SCREEN 3
PRINT STRING$("-",100)
WRAP OFF
PRINT STRING$("-",100)
```

**Remarks:** -

**See**

**Also:** SCROLL ON/OFF

## WRITE Command

**Action:** saves data to sequential files for later read with INPUT #.

**Syntax:** WRITE [#n],a[,a\$,b,...]  
*n: iexp, channel*  
*a,b: aexp*  
*a\$: sexp*

**Abbreviation:** wr [#n],a[...]

**Explanation:** The WRITE [#n] command is followed by numerical and string expressions which must be separated by commas. WRITE #n writes these expressions sequentially. The characters are enclosed in quotation marks and commas are generally used as separators.

**Example:** OPEN "o",#1,"A:\TEST.DAT"  
WRITE #1, 2\*PI,"Hello GFA",SIN(PI^2/4)  
CLOSE #1

**Remarks:** -

**See**  
**Also:** PRINT #, INPUT #



### XLATE\$() Function

**Action:** replaces all characters of a string expression with values from a table.

**Syntax:** XLATE\$(a\$,m())  
*a\$:* sexp  
*m():* integer variable (%,&,|) array

**Abbreviation:** -

**Explanation:** XLATE\$(a\$,m|()) converts the string expression a\$ using the user-created table in m|().

**Example:**

```
DIM m|(255)
FOR i%=32 TO 255
    m|(i%)=i%
NEXT i%
FOR i%=0 TO 31
    m|(i%)=46
NEXT i%
m|(155)=46
OPEN "I",#1, "TEST.DAT"
PRINT XLATE$(INPUT$(64,#1),m|())
CLOSE #1
```

```
// Prints 64 bytes from the file "TEST:DAT" and
// replaces all control characters and CHR$(155) with
// dots.
```

**Remarks:** XLATE\$(a\$,m%()) corresponds to

```
FOR i%=1 TO LEN(a$)
    MID$(a$,i%)=CHR$(m%(ASC(MID$(a$,i%,1))))
NEXT i%
```



**See**

**Also:**

UPPER\$(), LOWER\$(), UCASE\$(), LCASE\$()

### XOR() Function

**Action:** performs an exclusive bit-wise OR on two bit patterns.

**Syntax:** XOR(*i,j*)  
*i,j: iexp*

**Explanation:** XOR(*i,j*) sets only the bits which are set in one - and only one - of the two operands.

**Example:**

```
PRINT BIN$(3,4)           // prints 0011
PRINT BIN$(10,4)          // prints 1010
PRINT BIN$(XOR(3,10),4)    // prints 1001
```

**Remarks:** -

**See**

**Also:** AND(), OR(), IMP(), EQV()



# Appendix A



## The GFA-BASIC editor

### 1. General

The GFA-BASIC editor is a program editor written especially for the development of GFA-BASIC programs. It is a line oriented editor, in that it performs a syntax check for each line, it automatically indents the loops and subroutines and allows the usage of command abbreviations of GFA-BASIC commands.

The syntax check means that the editor tests if the given expression is syntactically correct for GFA-BASIC. If it isn't, a warning bell is sounded and the "Syntax error" message appears.

**Example:**                    prozedure test

In the above example the cursor remains on the line until the statement is either corrected or commented out. To comment a line out enter a double slash "/" at the beginning of the line. The correction for the above example reads

procedure test

If a statement is entered syntactically correct, GFA-BASIC editor converts all characters in GFA-BASIC commands and functions to uppercase. Therefore, GFA-BASIC creates the following out of the above statement

PROCEDURE test.

Indentation means that, in order to achieve better program structure, the GFA-BASIC editor automatically indents the contents of loops and sub-routines.

**Example:**

```
FOR i%=1 TO 100
  PRINT i%
NEXT i%
```

Abbreviations are allowed for nearly all GFA-BASIC commands. For example

```
f i% 1 100      FOR i%= 1 TO 100
p i%            PRINT i%
n i%            NEXT i%
```

In principle each line may only contain one command, but several statements are allowed if separated by commas.

**Example:**

```
FOR i%=1 TO 100
  a=7*i%,b=3.14*i%,c=PI*i%
NEXT i%
```

A program line can be up to 255 characters long. The comments are allowed either in between the statements or at the end of a statement. To separate a statement from a comment, the characters "/\*" (slash asterisk) are placed at the beginning and "\*/" (asterisk slash) at the end of the comment.

**Example:**

```
e%=a% /* start value */ * z% /* end value
```

If a GFA-BASIC statement is followed only by a comment it must be indicated by /\* or //.

**Example:**                    e%=a%\*z%                    // start value \* end value

or

                             e%=a%\*z%                    /\* start value \* end value

## 2.        The cursor key block block

When GFA-BASIC interpreter is run the GFA-BASIC editor is loaded. A blinking underdash, the cursor, appears below the menu bar. The cursor is controlled with the cursor keys as follows:

<b>Arrow left</b>	->	move cursor one character left
<b>Arrow right</b>	->	move cursor one character right
<b>Arrow up</b>	->	move cursor one line up
<b>Arrow down</b>	->	move cursor one line down
<b>End</b>	->	move cursor to line end
<b>Home</b>	->	move cursor to line start
<b>Pg up</b>	->	scroll one page up
<b>Pg down</b>	->	scroll one page down
<b>Ctrl + End</b>	->	move cursor to end of file
<b>Ctrl + Home</b>	->	move cursor to start of file
<b>Insert</b>	->	If there were no changes on the current line (the line with the cursor) a blank line is inserted above the current line. Otherwise, if the insert mode is on (menu bar F8) a space is inserted at the current cursor position.



**Delete**                    ->    Deletes the character under cursor and moves the remainder of the line left.

The cursor can also be moved by using a mouse. To do this, point the mouse pointer to the desired position and press the left mouse button.

### **3.        The numeric keypad**

When the Num(Lock) is off, the GFA-BASIC editor functions associated with the cursor key block can also be invoked from the numeric keypad.

### **4.        Control commands**

By combining the Control key (Ctrl, ^) and a letter a whole range of editor functions can be invoked without having to open the corresponding menu first. Furthermore, additional functions which are not implemented in the menu can also be invoked. The key assignment is as follows:

**Ctrl + Del**                ->    deletes the line with the cursor.

**Ctrl + Y**                   ->    deletes the line with the cursor.

**Ctrl + U**                   ->    Performs an "undelete". The last deleted line is inserted back in the text at the current position. After Ctrl + U the line which was brought back remains in the internal buffer. In this way, after a Ctrl + Y, the function Ctrl + U can be invoked repeatedly to perform a primitive copy of the deleted line.

**Ctrl + P**                   ->    Deletes the line from the current cursor position to the end of line.

- Ctrl + O**           -> Inserts, at the current cursor position, the last portion of the line previously deleted with Ctrl + P.
- Ctrl + N**           -> Inserts a blank line above the line with the cursor. In contrast to Insert, this function works even when there are changes on the current line.
- Ctrl + B**           -> Set the block start marker.
- Ctrl + K**           -> Sets the block end marker.
- Ctrl + R**           -> Scrolls one page down (Page down).
- Ctrl + C**           -> Scrolls one page up (Page up).
- Ctrl + E**           -> Replace text. To replace text, Alt + E must be used first to specify the search and replace strings.
- Ctrl + F or**  
**Ctrl + L**           -> Search text. Alt + L must be used before searching to specify the search string.
- Ctrl + Z**           -> Jump to the end of file (like Ctrl + Page/Down).
- Ctrl + Tab**          -> Jump one tab position left.
- Ctrl + G**           -> Opens the line number field for entry of a line number and jumps to the specified line.

The GFA-BASIC editor markers are set with **Ctrl + Function keys (F1,..., F10)**, and jumped to with **Alt + Function keys (F1,..., F10)**. These markers are just positions in the text and are not the markers for the program to jump to with **RESTORE** or **GOTO**.

The following markers are the defaults:

- |                  |    |  |
|------------------|----|--|
| <b>Alt + F7</b>  | -> | Jumps to the last cursor position before the switch to direct mode or before the last program run.   |
| <b>Alt + F8</b>  | -> | Jumps back to the position where the cursor was before the program was invoked from the editor. For example, when the program results in an error and is thereby terminated, the cursor is positioned to the line which caused the error. As a rule, this line is not the same as the one where the cursor was before the program run. By pressing Alt + F8 the cursor is then moved back to this original position. |
| <b>Alt + F9</b>  | -> | Jumps to the position from which the last search was initiated.  |
| <b>Alt + F10</b> | -> | Jumps to the last cursor position where a change was made.   |

## **5. Alt commands and menu bar**

Whenever the GFA-BASIC is run the GFA-BASIC editor is also loaded. The menu bar is situated at the upper edge of the screen. The first three entries (File, Search and Block) are real menus, since when they are activated a pull-down menu with various options "unfolds". The remaining entries in the menu bar are switches. In addition to the menu bar the GFA-BASIC editor also provides an info line (bottom line), which contains the current program file name and/or in case of an error the relevant error message.

The Alt commands serve mainly - in addition to the mouse - for opening of the menu bar of the GFA-BASIC editor. The individual menus can in most cases be activated with Alt + first letter of the menu title (with the excep-

tion of the switches Page Up and Page Down which can only be activated with the corresponding keys in the cursor block).

The entries in the first three menus can be invoked, assuming the menu is active (i.e. the pull-down menu has "unfolded"), by entering the letter in front of it (written in colour on colour monitors or with light letters on the monochrome monitor). The selection of individual menu entries can furthermore be performed in two additional ways:

- (1) Move the mouse pointer to the corresponding menu title and press the left mouse button. When the pull-down menu appears move the mouse pointer to the desired entry and press the left mouse button again.
- (2) To activate the first menu press the function key F1, to activate the second menu the function key F2 ... Move then the selection bar with the cursor keys (cursor key block) up or down, until the bar is on the desired entry. By pressing the Return key the corresponding entry is then selected. Optionally, you can activate any menu title with a function key and then and then "walk" left or right with the left and right cursor keys until the desired menu is active. The selection of the menu entry is then performed as already described. These three alternative ways of menu selection can be combined at will.

**Alt + F or F1**      ->      Opens the **File** menu.

This menu has four categories:

**Load, Save and  
Save as**

Loads or saves the program files in GFA-BASIC token format.

**Merge, Write and  
Print**

Loads or saves the program file in ASCII format and prints the file currently in the GFA-BASIC editor to the printer.

**New**

Deletes the program currently in the GFA-BASIC editor.

**Exit**

Leaves the GFA-BASIC editor and returns to the calling program.

**L**      ->      Selects the **Load** entry. This invokes a File-Select-Box. It contains the files in the current path. By typing in a file name or by selecting one with the scroll bars and the Return key, the file can be loaded into the GFA-BASIC editor. If the selected file is not in the GFA-BASIC token format an error message appears. **Load** can only load program files saved in the GFA-BASIC format (default extension ".GFA").

**S**      ->      Selects the **Save** entry. This saves the program currently in the editor, in GFA-BASIC format, under the given name in the current path. If the file was not previously loaded it is saved with the default name TEST.GFA.

- A** -> Selects the **Save as...** entry. The message "Save as TEST.GFA" appears on the info line of the GFA-BASIC editor. If you wish to save the program under a different name, TEST.GFA must first be deleted and the new name entered instead. It is also possible to enter a new path in front of the file name. Pressing the Return key then saves the file in GFA-BASIC format under the given name either in the old or in the new path.
- M** -> Selects the **Merge** entry. This invokes a File-Select-Box. It contains the files in the current path. By typing in a file name or by selecting one with the scroll bars and the Return key, the file can be loaded into the GFA-BASIC editor. The cursor line as well as all program lines below the cursor are thereby pushed "down". If the selected file is not in ASCII an error message appears.  
**Merge** can only load files in ASCII (default extension ".LST").
- W** -> Selects the **Write** entry. This saves the program currently in the editor, in ASCII, under the given name in the current path. If the file was not previously loaded it is saved with the default name TEST.LST.
- P** -> Selects the **Print** entry. A confirmation is required if the program currently in the GFA-BASIC editor is to be sent to the printer. If confirmed the program file is printed out.

- N** -> Selects the **New** entry. A confirmation is required if the program currently in the GFA-BASIC editor is to be deleted. If confirmed the program is deleted.
- X** -> Selects the **Exit** entry. A confirmation is required if the GFA-BASIC is to be terminated. If confirmed the control returns to the calling program.
- Alt + S or F2** -> Opens the **Search** menu.
- F** -> Selects the **Find** entry. A prompt for the search string appears next. Pressing the Return key makes the cursor jump from the current position to the first string which contains the search string. When no search string is specified the search function is cancelled.
- I** -> Selects the **Find Next** entry. Searches from the current cursor position for the next string which contains the search string specified with Alt + S + F. An alternative to Alt + S + I is also Ctrl + F (see above).
- E** -> Selects the **Exchange** entry. A prompt for the search and replace strings appears next. Pressing the Return key makes the cursor jump from the current position to the first string which contains the search string. The string is replaced only when Alt + S + X or Ctrl + E are pressed (see below).

- X** -> Selects the **Exchange Next** entry. Searches from the current cursor position for the next string which contains the string specified with Alt + S + E. Repeated pressing of the key combination Alt + S + X then replaces the search string with the replace string. As an alternative to Alt + S + X the key combination Ctrl + E can be used also.
- Alt + B or F3** -> Opens the **Block** menu.
- B** -> Selects the **Set Block (Begin)** entry. This marks the current cursor line as the first line in the block. Ctrl + B (see above) can also be used as an alternative to Alt + B + B.
- K** -> Selects the **Set Block (End)** entry. This marks the current cursor line as the last line in the block. Ctrl + K (see above) can also be used as an alternative to Alt + B + K.
- C** -> Selects the **Copy** entry. If a block is marked, it is copied to the current cursor position. The block markers remain, i.e. a block can be copied several times.
- M** -> Selects the **Move** entry. If a block is marked, it is moved to the current cursor position. The block markers are not deleted after the move.
- W** -> Selects the **Write** entry. If a block is marked a File-Select-Box appears, so that the name (and optionally the path) can be entered, under which the block is to be saved in ASCII.



- P** -> Selects the **Print** entry. Assuming a block is marked, the confirmation is required if the marked block is to be sent to the printer. If confirmed the block is printed out.
- H** -> Selects the **Hide** entry. Deletes the block markers.
- D** -> Selects the **Delete** entry. Assuming a block is marked, the confirmation is required if the marked block is to be deleted. If confirmed the corresponding block is deleted.
- Alt + D or F4** -> Switches to **Direct mode**. The GFA-BASIC commands entered in this mode are executed immediately after the Return key is pressed. To execute several statement lines a **PROCEDURE** or a **FUNCTION** can be written in the editor and then invoked in direct mode. The direct mode can be terminated by typing the GFA-BASIC command **EDIT**.
- Pg up or F5** -> **Scrolls one page up** (Page up).  
**Pg down or F6** -> **Scrolls one page down** (Page down).
- Alt + U or F7** -> **Undo**. Calling this function reverses all changes made on the current line. This can be done as long as the cursor stays on the same line.
- Alt + I or  
Alt + O or F8** -> Toggles between **Insert** and **Overwrite** modes.
- Alt + V or F9** -> Invokes the **View** function. This function shows the screen the way it was before returning to the GFA-BASIC editor.

- Alt + R or F10**      ->    Invokes the **Run** function. The program currently in GFA-BASIC editor is executed.
- Alt + T or  
Shift + F10**      ->    Invokes the **Test** function. Performs the test of the program structure, i.e. it test if all loops, subroutines and conditional statements of the program currently in the GFA-BASIC editor are complete.

The right side of the GFA-BASIC editor menu bar contains the line number of the line with the cursor. By pressing **Ctrl + G** or clicking with the mouse a line number can be entered in this place. The cursor then jumps to this line.

A unique feature of the GFA-BASIC editor is the ability to fold whole PROCEDURES and/or FUNCTIONS. The contents of these subroutines are then shown only by the header line of the PROCEDURE or FUNCTION. To indicate that this is a folded PROCEDURE or FUNCTION the header line is prefixed with a greater than character ">". To fold a PROCEDURE or FUNCTION move the cursor to the header line of the corresponding subroutine. The following can then be performed:

- Alt + Q**      ->    **Folds** the PROCEDURE or FUNCTION, whose header line contains the cursor. By pressing the key combination **Alt + Q** again the corresponding PROCEDURE or FUNCTION is unfolded.
- Alt + W**      ->    **Folds** all PROCEDURES and FUNCTIONS from the current cursor position. By pressing the key combination **Alt + W** again the corresponding PROCEDURES and FUNCTIONS are unfolded.



# **Appendix B**



# Keyboard Scan Codes

01	~68 ^5E ↑54 3B	~69 ^5F ↑55 3C	~6A ^60 ↑56 3D	~6B ^61 ↑57 3E	~6C ^62 ↑58 3F	~6D ^63 ↑59 40	~6E ^64 ↑5A 41	~6F ^65 ↑5B 42	~70 ^66 ↑5C 43	~71 ^67 ↑5D 44	~8B* ^89* ↑87* 85*	~8C* ^8A* ↑88* 86*			

~29* 29	~78 02	~79 03	~7A 04	~7B 05	~7D 06	~7C 07	~7E 08	~7F 09	~80 0A	~81 0B	~82 0C	~83 0D	~0E* 0E			
~A5* ^94* 0F	10	11	12	13	14	15	16	17	18	19	~1A* 1A	~1B* 1B				
		1E	1F	20	21	22	23	24	25	26	~27* 27	~28* 28	~2B* 2B	1C		
		*56	2C	2D	2E	2F	30	31	32	~33* 33	~34* 34	~35* 35				
						39										





















































## Note:

~ is equivalent to the Alt key

^ is equivalent to the Control key

↑ is equivalent to the Shift key

\* only to be used with INTR(\$16, \_AH=16)

\*56 only on German keyboard

## Code Table: Scan Codes

Hex		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
	Dec	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	↑ Tab
1	16	Alt Q	Alt W	Alt E	Alt R	Alt T	Alt Y	Alt U	Alt I	Alt O	Alt P	-	-	-	-	Alt A	Alt S
2	32	Alt D	Alt F	Alt G	Alt H	Alt J	Alt K	Alt L	-	-	-	-	-	Alt Z	Alt X	Alt C	Alt V
3	48	Alt B	Alt N	Alt M	-	-	-	-	-	-	-	-	F1	F2	F3	F4	F5
4	64	F6	F7	F8	F9	F10	-	-	Pos 1	↑	PgUp	-	←	-	→	-	End
5	80	↓	PgDn	Ins	Del	↑ F1	↑ F2	↑ F3	↑ F4	↑ F5	↑ F6	↑ F7	↑ F8	↑ F9	↑ F10	^ F1	^ F2
6	96	^ F3	^ F4	^ F5	^ F6	^ F7	^ F8	^ F9	^ F10	Alt F1	Alt F2	Alt F3	Alt F4	Alt F5	Alt F6	Alt F7	Alt F8
7	112	Alt F9	Alt F10	^ Prt	^ ←	^ →	^ End	^ PgDn	^ Pos1	Alt 1	Alt 2	Alt 3	Alt 4	Alt 5	Alt 6	Alt 7	Alt 8
8	128	Alt 9	Alt 0	Alt -	Alt =	^ PgUp											

### Note:

The key combinations **Alt 1** to **Alt 0** are composed of the Alt key plus a digit from the **top** row.

The key combination ↑[key] means Shift plus [key].

The key combination ^[key] means Control plus [key].

# Code-Table: ASCII Character Set

Dec	Hex	Character	Dec	Hex	Character	Dec	Hex	Character	Dec	Hex	Character
0	00	None	32	20	Space	64	40	@	96	60	'
1	01	^A*	33	21	!	65	41	A	97	61	a
2	02	^B*	34	22	"	66	42	B	98	62	b
3	03	^C*	35	23	#	67	43	C	99	63	c
4	04	^D*	36	24	\$	68	44	D	100	64	d
5	05	^E*	37	25	%	69	45	E	101	65	e
6	06	^F*	38	26	&	70	46	F	102	66	f
7	07	^G*	39	27	'	71	47	G	103	67	g
8	08	^H*	40	28	(	72	48	H	104	68	h
9	09	^I*	41	29	)	73	49	I	105	69	i
10	0A	^J*	42	2A	*	74	4A	J	106	6A	j
11	0B	^K*	43	2B	+	75	4B	K	107	6B	k
12	0C	^L*	44	2C	,	76	4C	L	108	6C	l
13	0D	^M*	45	2D	-	77	4D	M	109	6D	m
14	0E	^N*	46	2E	.	78	4E	N	110	6E	n
15	0F	^O*	47	2F	/	79	4F	O	111	6F	o
16	10	^P*	48	30	0	80	50	P	112	70	p
17	11	^Q*	49	31	1	81	51	Q	113	71	q
18	12	^R*	50	32	2	82	52	R	114	72	r
19	13	^S*	51	33	3	83	53	S	115	73	s
20	14	^T*	52	34	4	84	54	T	116	74	t
21	15	^U*	53	35	5	85	55	U	117	75	u
22	16	^V*	54	36	6	86	56	V	118	76	v
23	17	^W*	55	37	7	87	57	W	119	77	w
24	18	^X*	56	38	8	88	58	X	120	78	x
25	19	^Y*	57	39	9	89	59	Y	121	79	y
26	1A	^Z*	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[*	59	3B	;	91	5B	[	123	7B	{
28	1C	^\*	60	3C	<	92	5C	\	124	7C	
29	1D	]*	61	3D	=	93	5D	]	125	7D	}
30	1E	^^*	62	3E	>	94	5E	^	126	7E	~
31	1F	^_*	63	3F	?	95	5F	_	127	7F	

## Note:

IBM character set doesn't use all ASCII characters. For example, the first 32 characters are used as graphic characters.

\* Asterisk indicates the unprintable ASCII control characters. Most systems will display these characters as shown in this table. To display these characters you can normally use the given control key combinations.

from: IBM PC/AT Technical Reference



# Code Table: IBM ASCII Character Set

Dec	Hex	Char.	Dec	Hex	Char.	Dec	Hex	Char.	Dec	Hex	Char.
0	00	Space	38	26	&	76	4C	L	114	72	r
1	01	☺	39	27	'	77	4D	M	115	73	s
2	02	☹	40	28	(	78	4E	N	116	74	t
3	03	♥	41	29	)	79	4F	O	117	75	u
4	04	♦	42	2A	*	80	50	P	118	76	v
5	05	♣	43	2B	+	81	51	Q	119	77	w
6	06	♠	44	2C	,	82	52	R	120	78	x
7	07	•	45	2D	-	83	53	S	121	79	y
8	08	■	46	2E	.	84	54	T	122	7A	z
9	09	○	47	2F	/	85	55	U	123	7B	{
10	0A	◼	48	30	0	86	56	V	124	7C	
11	0B	♂	49	31	1	87	57	W	125	7D	}
12	0C	♀	50	32	2	88	58	X	126	7E	~
13	0D	🎵	51	33	3	89	59	Y	127	7F	△
14	0E	🎵	52	34	4	90	5A	Z	128	80	Ç
15	0F	☼	53	35	5	91	5B	[	129	81	ù
16	10	▼	54	36	6	92	5C	\	130	82	é
17	11	▲	55	37	7	93	5D	]	131	83	â
18	12	↕	56	38	8	94	5E	^	132	84	ä
19	13	≡	57	39	9	95	5F	_	133	85	à
20	14	☛	58	3A	:	96	60	`	134	86	á
21	15	§	59	3B	;	97	61	a	135	87	ç
22	16	¶	60	3C	<	98	62	b	136	88	ê
23	17	↕	61	3D	=	99	63	c	137	89	ë
24	18	↗	62	3E	>	100	64	d	138	8A	è
25	19	↓	63	3F	?	101	65	e	139	8B	ï
26	1A	→	64	40	@	102	66	f	140	8C	î
27	1B	←	65	41	A	103	67	g	141	8D	ì
28	1C	└	66	42	B	104	68	h	142	8E	Ä
29	1D	↔	67	43	C	105	69	i	143	8F	Å
30	1E	▲	68	44	D	106	6A	j	144	90	É
31	1F	▼	69	45	E	107	6B	k	145	91	æ
32	20	Space	70	46	F	108	6C	l	146	92	Æ
33	21	!	71	47	G	109	6D	m	147	93	ô
34	22	"	72	48	H	110	6E	n	148	94	ö
35	23	#	73	49	I	111	6F	o	149	95	ò
36	24	\$	74	4A	J	112	70	p	150	96	û
37	25	%	75	4B	K	113	71	q	151	97	ù

# **Appendix C**



Dec Hex Char.	Dec Hex Char.	Dec Hex Char.	Dec Hex Char.
152 98 ÿ	178 B2 █	204 CC 𐀀	230 E6 μ
153 99 Ö	179 B3 ı	205 CD ==	231 E7 τ
154 9A Ü	180 B4 𐀀	206 CE 𐀁	232 E8 Φ
155 9B ø	181 B5 𐀁	207 CF 𐀂	233 E9 Θ
156 9C £	182 B6 𐀂	208 D0 𐀃	234 EA Ω
157 9D ¥	183 B7 𐀃	209 D1 𐀄	235 EB δ
158 9E R	184 B8 𐀄	210 D2 𐀅	236 EC ∞
159 9F f	185 B9 𐀅	211 D3 𐀆	237 ED Ø
160 A0 á	186 BA 𐀆	212 D4 𐀇	238 EE ε
161 A1 í	187 BB 𐀇	213 D5 𐀈	239 EF ∩
162 A2 ó	188 BC 𐀈	214 D6 𐀉	240 F0 ≡
163 A3 ú	189 BD 𐀉	215 D7 𐀊	241 F1 ±
164 A4 ñ	190 BE 𐀊	216 D8 𐀋	242 F2 ≥
165 A5 Ñ	191 BF 𐀋	217 D9 𐀌	243 F3 ≤
166 A6 º	192 C0 𐀌	218 DA 𐀍	244 F4 ∫
167 A7 °	193 C1 𐀍	219 DB █	245 F5 ∫
168 A8 ı	194 C2 𐀎	220 DC █	246 F6 +
169 A9	195 C3 𐀎	221 DD █	247 F7 ≈
170 AA	196 C4 𐀏	222 DE █	248 F8 °
171 AB 1/2	197 C5 𐀏	223 DF █	249 F9 .
172 AC 1/4	198 C6 𐀐	224 E0 α	250 FA •
173 AD i	199 C7 𐀐	225 E1 β	251 FB √
174 AE «	200 C8 𐀑	226 E2 Γ	252 FC ²
175 AF »	201 C9 𐀑	227 E3 π	253 FD ²
176 B0 █	202 CA 𐀒	228 E4 Σ	254 FE █
177 B1 █	203 CB 𐀒	229 E5 σ	255 EF Space

Reference: IBM PC/AT Technical Reference, pages C-12, 13

# Graphic Characters

—					
196			179		
┌	┐	└	┌	┐	└
218	194	191	218	194	191
└	┐	┌	└	┐	┌
195	197	180	195	197	180
└	┐	┌	└	┐	┌
192	193	217	192	193	217

=					
205			179		
┌	┐	└	┌	┐	└
213	209	184	213	209	184
└	┐	┌	└	┐	┌
198	216	181	198	216	181
└	┐	┌	└	┐	┌
212	207	190	212	207	190

—					
196			186		
┌	┐	└	┌	┐	└
214	210	183	214	210	183
└	┐	┌	└	┐	┌
199	215	182	199	215	182
└	┐	┌	└	┐	┌
211	208	189	211	208	189

=					
205			186		
┌	┐	└	┌	┐	└
201	203	187	201	203	187
└	┐	┌	└	┐	┌
204	206	185	204	206	185
└	┐	┌	└	┐	┌
200	202	188	200	202	188

Reference: IBM PC/XT Technical Reference, page C-13

# DOS Interrupts

Interrupt	Function	Description
\$10	0	VIDEO - define video mode
	1	VIDEO - define cursor type
	2	VIDEO - set cursor position
	3	VIDEO - get cursor position
	4	VIDEO - returns lightpen position
	5	VIDEO - select screen page
	6	VIDEO - scroll screen up
	7	VIDEO - scroll screen down
	8	VIDEO - read character with attribute
	9	VIDEO - write character with attribute
	0A	VIDEO - write character at cursor position
	0B	VIDEO - set colour palette
	0C	VIDEO - set pixel
	0D	VIDEO - get pixel
	0E	VIDEO - write text in teletype mode
	0F	VIDEO - returns current video mode
	10	VIDEO - set palette register
	11	VIDEO - character generator
	12	VIDEO - function call
	13	VIDEO - write string
	14	VIDEO - load LCD character set
	15	VIDEO - return physical parameter
	16	RESERVED
	17	RESERVED
	18	RESERVED
	19	RESERVED
	1A	VIDEO - read/write hardware configuration
	1B	VIDEO - return video status
	1C	VIDEO - save/restore VIDEO status
	1D-FF	RESERVED
\$11	-	return system configuration
\$12	-	memory size

# DOS Interrupts

Interrupt	Function	Description
\$13	0	DISK DRIVE - system initialization
	1	DISK DRIVE - read status
	2	DISK DRIVE - read from disk
	3	DISK DRIVE - write to disk
	4	DISK DRIVE - test sectors
	5	DISK DRIVE - format track
	6	DRIVE - format cylinder and set bad sector flags
	7	DRIVE - format drive and start with cylinder
	8	DRIVE - return current drive parameters
	9	DRIVE - install drive
	0A	DRIVE - read extended sectors
	0B	DRIVE - write extended sectors
	0C	DRIVE - search for cylinder
	0D	DRIVE - additional disk reset
	0E	DRIVE - read sector buffer
	0F	DRIVE - write sector buffer
	10	DRIVE - test if drive is ready
	11	DRIVE - park heads
	12	DRIVE - RAM test controller
	13	DRIVE - drive test
	14	DRIVE - controller test
	15	DRIVE - read DASD type
	16	DISK DRIVE - return disk change status
	17	DISK DRIVE - set DASD type for formatting
	18	DISK DRIVE - set media type
	19	DRIVE - park heads
	1A	DRIVE - format unit
	1B-FF	RESERVED
\$14	0	SERIAL - initialize communication port
	1	SERIAL - send character to serial port
	2	SERIAL - receive character from serial port
	3	SERIAL - port status

# DOS Interrupts

Interrupt	Function	Description
\$14	4	SERIAL - extended initialization
	5	SERIAL - extended port control
	6-FF	RESERVED
\$15	0	CASSETTE - motor on
	1	CASSETTE - motor off
	2	CASSETTE - read data block
	3	CASSETTE - write data block
	4-0E	RESERVED
	0F	DRIVE - periodic formatting
	10-1F	RESERVED
	20	AL=10 SYSREQ on AL=11 SYSREQ off
	21	DEVICE - self test error list
	22-3F	RESERVED
	40	DEVICE - read/test (profile)
	41	DEVICE - wait for external event
	42	DEVICE - system power supply OFF (Request)
	43	DEVICE - read system status
	44	DEVICE - activate internal modem power supply
	45-4E	RESERVED
	4F	KEYBOARD - capture keyboard
	50-7F	RESERVED
	80	DEVICE - open device
	81	DEVICE - close device
	82	DEVICE - terminate program
	83	DEVICE - interval wait
	84	JOYSTICK
	85	SYSTEM - System Request key
	86	DEVICE - wait
	87	DEVICE - move block
	88	MEMORY - return size of extended memory
	89	MEMORY - switch to protected mode



# DOS Interrupts

Interrupt	Function	Description
\$15	90	DEVICE - device busy
	91	DEVICE - turn interrupt off and set flag
	92-BF	RESERVED
	C0	DEVICE - return system parameters
	C1	DEVICE - segment address of extended BIOS
	C2	DEVICE - BIOS interface for handle-DEVICE
	C3	DEVICE - activate timeout watch-dog
	C4	DEVICE - programable random selection
	C5-FF	RESERVED
\$16	0	KEYBOARD - read characters from keyboard
	1	KEYBOARD - return keyboard status
	2	KEYBOARD - return keyboard flags
	3	KEYBOARD - delay
	4	KEYBOARD - key click ON/OFF
	5	KEYBOARD - write characters
	6-0F	RESERVED
	10	KEYBOARD - extended character read from keyboard
	11	KEYBOARD - extended key inquiry
	12	KEYBOARD - return extended shift status
	13-FF	RESERVED
\$17	0	PRINTER - send character to printer
	1	PRINTER - initialize printer port
	2	PRINTER - set printer status
	3-FF	RESERVED
\$18	-	BASIC - load Basic
\$19	-	BOOTSTRAP - load boot sector
\$1A	0	TIMER - read time
	1	TIMER - set time
	2	TIMER - get RTC clock
	3	TIMER - set RTC clock
	4	TIMER - get RTC date

# DOS Interrupts

Interrupt	Function	Description
\$1A	5	TIMER - set RTC date
	6	TIMER - set RTC alarm
	7	TIMER - delete RTC alarm
	8	TIMER - RTC activated power on
	9	TIMER - read RTC alarm and status
	0A	TIMER - get system counter (days)
	0B	TIMER - set system counter (days)
	0C-7F	RESERVED
	80	SOUND - multiplexer
	81-FF	RESERVED
\$21	0	terminate program
	1	read and output characters from keyboard
	2	display characters
	3	receive character from serial port
	4	send characters to serial port
	5	send characters to parallel port
	6	direct character I/O
	7	direct character input without echo
	8	read characters from keyboard
	9	print characters
	A	buffered string input
	B	get keyboard status
	C	delete input buffer and invoke input
	D	drive reset
	E	select drive
	F	open file
	10	close file
	11	search for first FCB entry
	12	search for next FCB entry
	13	delete file
	14	sequential read
	15	sequential write

# DOS Interrupts

Interrupt	Function	Description
\$21	16	create file
	17	rename file
	18	RESERVED
	19	return current drive
	1A	set DTA (Disk Transfer Address)
	1B	return allocation for current drive
	1C	return allocation for specific drive
	1D	RESERVED
	1E	RESERVED
	1F	RESERVED
	20	RESERVED
	21	random read
	22	random write
	23	get file size
	24	set record number
	25	set interrupt vector
	26	create new program segment
	27	random block read
	28	random block write
	29	parse filename
	2A	get date
	2B	set date
	2C	get time
	2D	set time
	2E	set verify flag
	2F	return DTA
	30	get DOS version number
	31	terminate program and stay resident
	32	RESERVED
	33	get/set Control-C status
	34	RESERVED
	35	get interrupt vector

# DOS Interrupts

Interrupt	Function	Description
\$21	36	get disk free space
	37	RESERVED
	38,0	get country dependent information
	38,X	set country dependent information (X=country code)
	39	create subdirectory
	3A	remove subdirectory
	3B	change directory
	3C	create file (handle)
	3D	open file (handle)
	3E	close file (handle)
	3F	read from file or device (handle)
	40	write to file or device (handle)
	41	delete file (handle)
	42	move file pointer
	43	get/set file attribute
	44,0	IOCTL - get device attribute
	44,1	IOCTL - set device attribute
	44,2	IOCTL - receive characters from driver
	44,3	IOCTL - send characters to driver
	44,4	IOCTL - receive character from block driver
	44,5	IOCTL - send characters to block driver
	44,6	IOCTL - return input status
	44,7	IOCTL - return output status
	44,8	IOCTL - is medium changeable?
	44,9	IOCTL - remote device test
	44,A	IOCTL - remote handle test
	44,B	IOCTL - change retry count
	44,C	IOCTL - handles
	44,D	IOCTL - devices
	44,E	get drive
	44,F	set drive
	45	duplicate a file handle

# DOS Interrupts

Interrupt	Function	Description
\$21	46	force same file handle
	47	get current directory
	48	allocate memory
	49	free allocated memory
	4A	set allocate memory block
	4B,0	program load and execute
	4B,3	load overlay
	4C	terminate process
	4D	return end code from a process
	4E	find first file entry
	4F	find next file entry
	50	RESERVED
	51	RESERVED
	52	RESERVED
	53	RESERVED
	54	return verify status
	55	RESERVED
	56	rename file
	57	get/set file time/date stamp
	58	get/set file recording mode
	59	get extended error
	5A	create temporary file
	5B	create new file
	5C,0	lock access
	5C,1	unlock access
	5D	RESERVED
	5E,0	return machine name
	5E,2	printer setting
	5F,2	read entry from allocation table
	5F,3	insert entry into allocation table
	5F,4	delete entry from allocation table
	60	RESERVED

# DOS Interrupts

Interrupt	Function	Description
\$21	61	RESERVED
	62	return PSP address
	63	get byte table
	65	read extended country dependent information
	66	get/set code page
	67	set handle count
	68	deliver file
\$24	-	critical error handler vector
\$25	-	read data from disk (absolute)
\$26	-	write data to disk (absolute)
\$33	-	mouse interrupt
\$67	-	EMS interrupt

Reference: Programmer's Guide to the IBM PC, (Microsoft Press), Peter Norton, chapter 8 to 13  
IBM PC/XT Technical Reference BIOS Listings  
IBM PC/AT Technical Reference BIOS Listings  
IBM PS/2 and PC BIOS Interface Technical Reference, pages 2-10 to 2-122  
IBM DOS 3.3 Technical Reference, pages 6-1 to 6-33 and 6-6 to 6-7



# Appendix D





# International sorting sequence

sorted by ASCII code values:

ASCII	Character	ASCII	Character	ASCII	Character
65	A	100	d	134	ä
66	B	101	e	135	ç
67	C	102	f	136	ê
68	D	103	g	137	ë
69	E	104	h	138	è
70	F	105	i	139	ï
71	G	106	j	140	î
72	H	107	k	141	ì
73	I	108	l	142	Ä
74	J	109	m	143	Å
75	K	110	n	144	É
76	L	111	o	145	æ
77	M	112	p	146	Æ
78	N	113	q	147	ô
79	O	114	r	148	ö
80	P	115	s	149	ò
81	Q	116	t	150	û
82	R	117	u	151	ù
83	S	118	v	152	ÿ
84	T	119	w	153	Ö
85	U	120	x	154	Ü
86	V	121	y	160	á
87	W	122	z	161	í
88	X	128	Ç	162	ó
89	Y	129	ü	163	ú
90	Z	130	é	164	ñ
97	a	131	â	165	Ñ
98	b	132	ä	225	ß
99	c	133	à		

# sorted alphabetically

ASCII	Character	ASCII	Character	ASCII	Character
97	a	139	ĩ	82	R
132	ä	161	í	115	s
160	á	141	ì	225	ß
133	à	140	î	83	S
131	â	73	I	116	t
65	A	106	j	84	T
142	Ä	74	J	117	u
98	b	107	k	129	ü
66	B	75	K	163	ú
99	c	108	l	151	ù
135	ç	76	L	150	û
67	C	109	m	85	U
128	Ç	77	M	154	Û
100	d	110	n	118	v
68	D	164	ñ	86	V
101	e	78	N	119	w
137	ë	165	Ñ	87	W
130	é	111	o	120	x
138	è	148	ö	88	X
136	ê	162	ó	121	y
69	E	149	ò	152	ÿ
144	É	147	ô	89	Y
102	f	79	O	122	z
70	F	153	Ö	90	Z
103	g	112	p	134	à
71	G	80	P	143	Å
104	h	113	q	145	æ
72	H	81	Q	146	Æ
105	i	114	r		

from: Paradox 2.0 User's Guide (US-Original!), pages 519 to 521

## Conversion: Hexadecimal to decimal

Word		Byte and Nibble		Byte		Nibble	
Hex	Dec	Hex	Dec	Hex	Dec	Hex	Dec
0000	0	000	0	00	0	0	0
1000	4096	100	256	10	16	1	1
2000	8192	200	512	20	32	2	2
3000	12288	300	768	30	48	3	3
4000	16384	400	1024	40	64	4	4
5000	20480	500	1280	50	80	5	5
6000	24576	600	1536	60	96	6	6
7000	28672	700	1792	70	112	7	7
8000	32768	800	2048	80	128	8	8
9000	36864	900	2304	90	144	9	9
A000	40960	A00	2560	A0	160	A	10
B000	45056	B00	2816	B0	176	B	11
C000	49152	C00	3072	C0	192	C	12
D000	53248	D00	3328	D0	208	D	13
E000	57344	E00	3584	E0	224	E	14
F000	61440	F00	3840	F0	240	F	15

### Note:

To convert a hexadecimal word (2 bytes) to decimal, find the decimal value for each hexadecimal digit and then add all decimal values up.

### Example:

The hexadecimal word A5D7 has the decimal value of 42455. A000H is equivalent to 40960, 500H is equivalent to 1280, D0H is equivalent to 208, 7H is equivalent to 7.

$40960 + 1280 + 208 + 7 = 42455$ .

# Conversion: hexadecimal - decimal

Byte value

		low value															
	HEX	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
high value	00	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	10	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
	20	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
	30	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
	40	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
	50	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
	60	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
	70	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
	80	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
	90	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
	A0	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
	B0	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
	C0	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
	D0	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
	E0	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
	F0	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Instructions:

For example, if you wish to determine the decimal value of hexadecimal E9, search the far left column for E0 (high nibble in E9 hex is E), and then locate the 9 on the top row. The value where the row and column intersect is the required decimal value.



GFA Systemtechnik GmbH  
Heerdter Sandberg 30  
D-4000 Düsseldorf 11  
Tel. 02 11/55 04-0  
Fax 02 11/55 04 44

